KUNGLIGA TEKNISKA HÖGSKOLAN (KTH)

Synchronization of Tasks in Multiprocessor Systems-on-Chip

José Henrique de Magalhães Simões Calado

Master in Electrical and Computers Engineering

Examiner: Associate Professor Ingo Sander

Advisor: Jun Zhu

Advisor: Professor José Carlos Alves

October 2010
TRITA-ICT-EX-2010:278

Abstract

Multiprocessor systems on chip (MPSoC) are getting more popular as new ways to take advantage of them are appearing. They are very useful, typically in embedded applications, since they can be tailored according to the requirements. These architectures meet the performance needs of several different types of applications such as, multimedia, telecommunication, and network security.

The aim of this work was to study the connection between tasks, either if they are inside the same processor, or outside and distributed on several different processors connected by buffers of first-in-first-out type.

We have used a Stratix II EP2S60 FPGA chip that, thanks to the Altera tools, has simplified the whole hardware prototyping part. Thus, we have mainly focused in the software running on Nios II soft-core processors, specially regarding the synchronization and communication inter-task.

In order to evaluate the performance, an abstract topology inspired on the Google's MapReduce framework was created. In a streaming data-flow it can be seen as two switching nodes; one that splits data among the workers processors, and another that joins the results and sends them to the next stage.

In our experiments we have used Nios II soft-core processors with tightly coupled memories and we have created a pseudo-random generator as a Nios II custom instruction in order to speedup the generation of numbers for a Monte Carlo test application. Several different approaches were taken into consideration and, for the generation of 50K random numbers, we have experienced a speed-up of 2.03x (19.21μs –> 9.43μs per random Cartesian point) between a sequential and a parallel version with 2 workers processors, using fast Nios II with tightly coupled memories.

Acknowledgments

I have to say that being on Sweden during one term was an incredible experience. One of the most important of my life so far. In the beginning I was not even expecting to go and suddenly everything happened really fast. I've met incredible people that have supported, helped and shared amazing new experiences with me. I want to thank everyone who, somehow, changed something important inside of me. Now I feel something that sounds better with the beautiful Portuguese word "Saudade". No direct translation exists but it means the *feeling* when someone misses something good.

However, during the journey, I always felt my roots of Porto, my hometown. Specially because of Silvia and her simple, and yet powerful, love and friendship. By living abroad I've learned to give more value to all the things that I care and to pay more attention to all the good things that are always happening.

I'm eternally grateful to my Parents for everything. They have been on my side no mater what, and never asked anything in return, only that I continued to seek the path to happiness. My brother and best friend, Guilherme, who always gave me the strength to continue even when things seemed to not go that well... Obrigado!

Finally, I will never forget the amazing coffees *et al.* with Ramon and Willeke, whose empathy has always encouraged me to pursuit my goals. Also, I would like to thanks all the people that I've crossed during these months in Sweden, specially the ones from "Jyllandsgatan".

Muito obrigado a todos os meus amigos em Portugal, em particular ao Filipe.

A special thanks to Jun for his patience, support, ideas and guidance through this thesis.

The Author

“You were right about the stars,
Each one is a setting sun.”

Jeff Tweedy

Contents

List of Figures

List of Tables

Abbreviations and Symbols

CI	Custom Instruction
CPU	Central Processor Unit
FEUP	Faculdade de Engenharia da Universidade do Porto
FIFO	First-In-First-Out (Buffer)
FPGA	Field-Programmable Gate Array
IDE	Integrated Development Environment
IP	Intelectual Property
KTH	Kungliga Tekniska högskolan
LE	Logic Elements
LFSR	Linear Feedback Shift Register
MIEEC	Mestrado Integrado em Engenharia Electrotécnica e de Computadores
MPSoC	Multiprocessor System-on-Chip
PCB	Printed Circuit Board
PRNG	Pseudo Random Number Generator
RTOS	Real-Time Operation System
SOPC	System on Programmable Chip
TCM	Tightly Coupled Memory
WCET	Worst Case Execution Time

Chapter 1

Introduction

Multiprocessor systems-on-chip (MPSoCs) are getting more popular due to the great advances in microelectronics that are currently taking place. The integration is getting better with lower power consumption. However, there is the need to develop ways to extend the natural boundaries of physics by developing new strategies since we need a bigger processing power and, in many situations, with lower consumptions.

The trend is to create smaller and more ecologic devices that, together in a network, can deliver much more computational power than as if they were working alone. Of course that the research starts from the simple problem and then, like in a *bottom-up* work flow, we are slowly creating "clouds" formed by regular computers with typical architectures, smart phones, PDAs, tablets, and several new emerging gadgets. This mesh is extremely vast and is expanding everyday.

Some of them have already MPSoCs inside; for example, many modern cell-phones have in the same chip the general-purpose CPU, GPU (Graphic Processing Unit), DSP and many more IP modules. The Nintendo Wii console combines a PowerPC 750CX and a GPU in the same chip, hence it is also a heterogeneous MPSoC [1].

When reducing this "cloud" perspective to the *chip level*, it is possible to see a heterogeneous multiprocessor SoC. Actually the clouding computation is also getting more popular because it delivers a higher processing mechanisms with lower costs, but security risks are still a big and real concern among the companies. This is beyond the scope of this thesis, though.

Common sense says that there is strength in numbers and that is valid for almost everything that handles a lot of information (of any kind). Obviously, using parallel methodologies it is possible to increase the amount of work done in less time. However, since this is a relatively new field of research, there are still several pieces that can, and will, be better evaluated to improve overall performance. One important, that will be the main theme of this thesis, is the communication between processors and between peripherals inside the same chip. The interconnect network is an important factor that limits the MPSoC performance and is as important as the capacity of the node processors (e.g. CPU speed, cache size, etc.) [2].

Many multiprocessor designs are based on shared resources due to their low cost, however the scalability is limited and the concurrency is penalized due to the masters competition in the access to the bus. In order to solve this, many studies are being made in order to have on-chip micro-networks (or Networks-on-Chips, NoCs) to interconnect all IP blocks [3]. Despite the fact that this is a topic beyond the scope, this work presents some ideas that can be helpful to improve their design. However, our target is to insert small data-flow models into small embedded systems, and find ways to efficiently interconnect the nodes.

1.1 Motivation and Objectives

Several works have been published regarding multiprocessor systems-on-chip. They are proposing various types of architectures and models of computation (MoCs) in order to fulfill the requirements for a specific application. However, a global framework that can be used by the most generic applications, with defined standards and rules is still in the childhood since there are a lot of different perspectives. A homogeneous designing flow could greatly improve the task of creating new platforms and corresponding software. For that, there is the need to improve and define the interconnection mechanisms and to find better methodologies to map and distribute tasks among processors.

A MoC consists of the definition of the set of allowable operations used in computation and their respective costs in the multiprocessor paradigm. Two widely used are Kahn process network (KPN) [4] and synchronous data flow (SDF) [5].

KPN describes an application as a network of autonomous processes that communicate through FIFO channels. It has clearly defined semantics and allows to explicitly express the parallelism through the separation of communication and computation. The main advantage is that it allows to avoid many pitfalls of parallel execution, such as data races, nondeterminism, or the need to strict synchronization [6].

This MoC is very useful for streaming applications, such as audio and video codecs, signal processing applications, or networking applications because their algorithms structures matches the KPN model. However, the communication between processes and their orchestration can limit the speed-up achievable by parallel execution.

A SDF MoC, when compared to the KPN, is more restrictive in the sense that there is a fixed token production and consumption rate. However, it allows to statically schedule an application to reduce the context switching overhead. In fact, the construction of periodic schedules at compilation time is possible with bounded memory [7].

Typically, the FIFOs are unbounded which leads to the impossibility of analysing the dimension and costs of memory modules [8]. To overcome this problem, a timed extension has been applied to the SDF MoC, but the single-unit time assumption [9, 10] is not useful for heterogeneous computation and communication timing in MPSoCs.

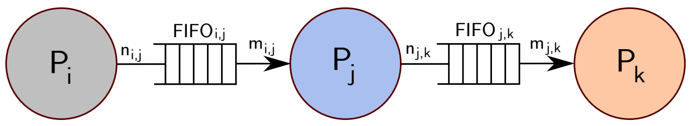


Figure 1.1: Streaming application model example

In this thesis we followed an alternative to program multiprocessor systems without restricting to a certain model of computation, in order to focus more on the synchronization and communication mechanisms. However, we started from a SDF application model with three-stage pipeline which was used as a tutorial example in [11] and is depicted in figure 1.1. The author explores the SDF semantics in order to minimize the buffer requirements without affecting the throughput for streaming applications.

The nodes are processes where computation occurs and the edges are the channels with finite buffers of type FIFO. As said before as one of the biggest restriction of SDF model, the input and output rates of the processes, or $m_{i,j}$ and $n_{i,j}$, are fixed.

We tried to respect the flow of SDF MoC with some differences regarding the semantics: Although in [10] the data tokens remain on the input side FIFO while the computation of a process is completed, in this work when a process is triggered, it reads and immediately consumes the tokens. The second difference is that we do not use any type of static scheduling policy, only a *best effort* dynamic scheduling. As soon as data is available in the input side of a node and there is space in the output side inside the FIFO, the process starts. This is not very efficient since the processors are always running at full-speed, but it allows to check the biggest possible performance achievable when using different types of synchronization. Furthermore, it can be easily modified to have a static scheduler, or a more complex dynamic scheduler, to became suitable for real-time streaming applications.

In some cases, when a streaming application is divided into several stages mapped on different processors, the biggest worst case execution time (WCET) of a specific stage dictates the throughput that, in a real-time environment, might not respect the deadlines. Therefore, our goal is to parallelize one or more pipeline stage(s) in order to reduce the overall WCET. For instance, assuming that in the figure 1.1 the biggest WCET belongs to P_j and it is 5 ms. The WCET of P_i and P_k is only 1 ms and, after the initial start-up delay, the real-time requirements dictates one new P_k conclusion every 3 ms. Ideally, if P_j can be divided in two sub-nodes, $P_{j,1}$ and $P_{j,2}$, where each one of them needs 5 ms to finish the job but, when combined, the WCET is 2.5 ms.

Of course that there are the mapping and synchronization delays, and most of the times the computation can not be completely parallelized. This was only an utopian scenario to show the idea. Actually, we were more interested to evaluate the buffers and synchronization than the mathematical formalisms of this approach.

This can be compared to how Google's *MapReduce* framework processes large data sets. A huge amount of data is splitted and mapped into several computers that work independently and, when they finish the processing, there is the reduction stage where the results are concatenated

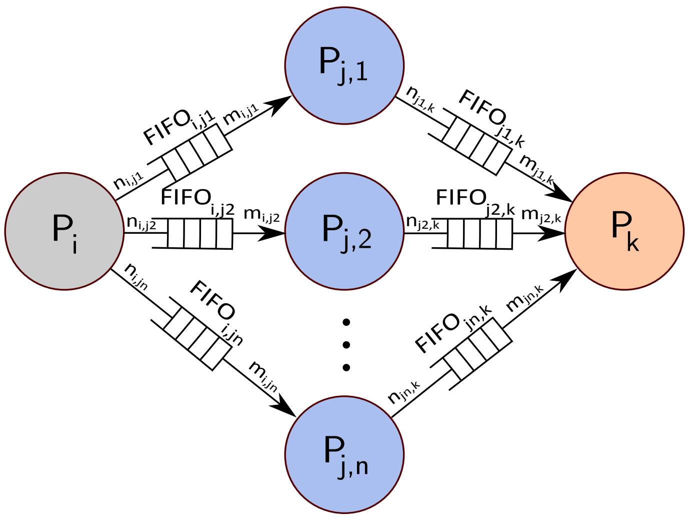


Figure 1.2: Simple data-flow model example

and sent to the output [12]. As shown in the figure 1.2, the node P_i, or *Source* Task from now on, is responsible for the distribution of the data into the workers. The node P_k, or *Destination* Task, is where the results obtained from the workers, $P_{j,x}$, are concatenated.

The authors in [13] presented two synchronization mechanisms for MPSoCs. The mailbox based synchronization has low latency and low resource overhead, but it is not feasible for a large number of processors because they share the same buses. On the other hand, the packet switch method has larger latency as it needs the headers to be processed in order to route them toward the destinations, but it has more scalability and feasibility.

Another goal of this thesis is to clearly describe the fundamentals behind the MPSoC concept in a higher level, since the hardware functionality is automatically created by the Altera tools, using the SOPC Builder. We just need to "say" which components we want and the corresponding connections. Therefore, we do not have to worry about the system level design, particularly regarding the power issues and the inherent complexity of the physical mesh between several processing units.

1.2 Contributions

The main contributions of this thesis are described next:

1. Present a brief study about the work done in the last years about MPSoCs and its challenges.

2. Elaborate several MPSoCs with Nios II cores with tightly coupled memories and custom hardware modules.

3. Contribute with a comparison between the most common ways to synchronize different CPUs using buffers that can be used in streaming or pipelined real-time applications.

4. Integrate a Monte Carlo method inside the MPSoC to extend the experiments regarding the communication between processors and to test the hardware random generator.

5. Develop a way to get a truly random seed for the random generator using a Nios II custom interruption module.

1.3 Document Layout

Besides the introduction, this thesis has four more chapters. In chapter 2, the board from Altera with the FPGA chip is described and introduces all the pieces used to form several MPSoC platforms. Also, it shows the reader, the architecture followed and introduces the basic software components like the real-time operative system. Chapter 3 presents some experiments with an abstract application that synchronize tasks and forces communication between processors. One particular case-study is debated in chapter 4, using the work of the previous chapter adding more concepts. Finally, chapter 5 concludes the thesis and proposes future work.

Chapter 2

FPGA based MPSoC

In this chapter there is an overview of all components and ideas used for this thesis that were performed together in several experiments in order to find interesting results.

2.1 Introduction

In order to fully understand all the following chapters, it is important to clearly describe all the components that were used. This chapter is divided in three sections where the first one aims the definition of the target hardware and the description of the main hardware modules used to form a system that, because it is inside the FPGA chip (SoC) and there are multiple processors running, it is a MPSoC (Multiprocessor System-on-Chip). The second section is about the chosen architecture for the MPSoC platform and how the components are connected. Finally, in the third section, the main software modules running on the Nios II processors are explained.

The possible optimizations during the software compilation are also explained here since it was a background study followed along the rest of this thesis. Also, a simpler SoC was made to test and prove the usability of tightly coupled memories as a useful storage place for data and instructions, even for the case where there is a lot of data transactions between processors.

2.2 Hardware

The hardware platform used is a Nios Development Board, which contains a Altera Stratix II EP2S60 FPGA chip.

The architecture of the Stratix II edition of Nios Development Board is depicted in the figure 2.1. As it is presented, this platform is comprised of a FPGA chip, communication interfaces, data buses, memories and I/O pins. The system on-chip is built inside the FPGA chip with one or more Nios II, which are 32-bit RISC processors embedded in resident logic (soft core), BRAM

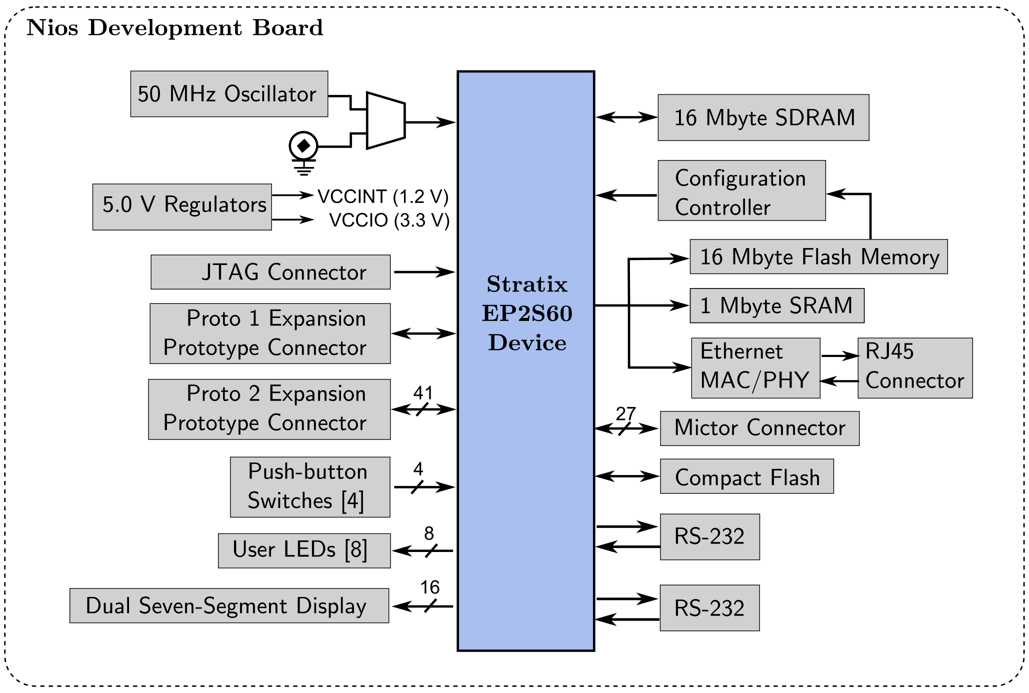


Figure 2.1: Block Diagram of Nios Development Board

memories, data buses and I/O controllers. This board is also supported by a developing environment from Altera, Quartus II.

The most relevant features of this board are as described, according to the reference manual [14]:

- **FPGA Chip** – Stratix II EP2S60F672C5ES device with 24,176 adaptive logic modules (ALM) and 2,544,192 bits of on-chip memory;
- **External Memory** – 16 Mbytes of flash memory, 1 Mbyte of static RAM, and 16 Mbytes of SDRAM;
- **Interface** – JTAG connectors to Altera® devices via Altera download cables;
- **Crystal Oscillator** – 50 MHz.

The letters “ES” on the FPGA device label means that it is an engineering sample. That is due to the fact that it came in early shipments of the Nios development board and it has a known issue affecting the M-RAM blocks. Thus, the two of them available in this device were not used, which means less 1,179,648 bits (144 kbytes) of on-chip memory.

The 50 MHz oscillator is socketed and can be changed or removed. There is the possibility to use the external clock connector but that would require to change the configuration controller. Despite of the fact that this FPGA device can work with a maximum theoretical clock of 550 MHz, all the experiments were made using the 50 MHz oscillator. A maximum frequency is usually computed during the system generation based on the distance of the biggest path and optimizations are made in order to respect the given clock rate.

BRAM (Block RAM) is comprised by three different memory block structures; the already spoken M-RAM not used in this thesis, M512 RAM and M4K RAM. M512 RAM blocks are simple dual-port memory blocks with 512 bits plus parity (576 bits) and M4K RAM blocks are true dual-port memory blocks with 4K bits plus parity (4,608 bits). This FPGA device has 329 blocks of M512 RAM and 255 blocks of M4K RAM, hence a total of 145 kbytes of on-chip memory available to the user.

The other important type of memory used in this thesis is the external PC100 Synchronous Dynamic RAM (SDRAM). It has 16 Mbytes and is meant to work at a speed of 100 MHz, but in this work it is clocked at the same speed as the rest of the system: 50 MHz.

These two types of memory are used to store data and instructions. However, the access to SDRAM is very slow when compared to the on-chip memory but has significant more space.

The on-board FPGA chip is a powerful system and its attributes are summarized by the following table:

Table 2.1: Summary of Stratix II EP2S60 specifications

ALMs	**ALUTs**	**Equivalent LEs**	**Total RAM bits**	**User I/O pins**
24,176	48,352	60,440	2,544,192	492

The adaptive logic module (ALM) is the basic building block of logic in the Stratix II architecture. Each ALM contains a variety of look-up table (LUT)-based resources that can be divided between two adaptive LUTs (ALUTs). This adaptability allows the ALM to be completely compatible with four-input LUT architectures, and makes it possible to have a equivalent counting of logic elements (LEs) [15].

Several different platforms were built in order to pursuit interesting results. In figure 2.5 it is possible to see a simple MPSoC with shared memory.

In the following sections is presented the way that everything works. They are the "pieces" of the final goal; which is the best way to synchronize different processors, or tasks, with the available modules. It will start with a brief description of Nios II soft-core processor, then how is it possible to use tightly coupled memories and custom instructions with this processor. The description of the fabric used to interconnect all the components is given in section 2.2.3. After, a brief explanation of how several hardware module works are given, namely the performance counter to measure time, the timer, the mailbox and finally the IRQ mailbox.

2.2.1 Nios II

Nios II is a soft-core processor provided by Altera [16]. It has an architecture of 32-bit RISC suitable for embedded applications and it is entirely implemented using the FPGA resources. One of the major advantages is his highly customizable features. There are three different types in size and features but the core remains the same and follows the Harvard architecture, where the data

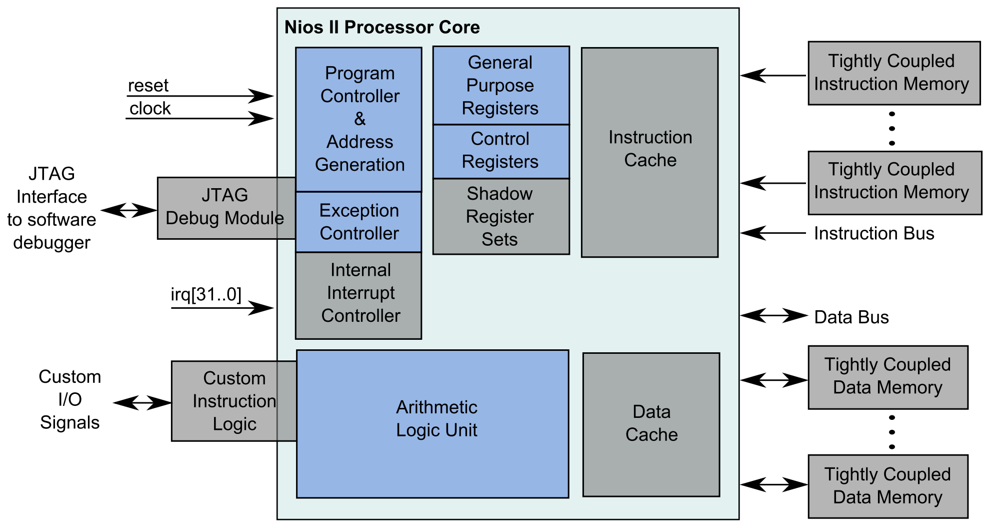

Nios II Processor Core
reset
clock
JTAG Interface to software debugger
JTAG Debug Module
irq[31..0]
Custom I/O Signals
Custom Instruction Logic
Program Controller & Address Generation
Exception Controller
Internal Interrupt Controller
General Purpose Registers
Control Registers
Shadow Register Sets
Instruction Cache
Arithmetic Logic Unit
Data Cache
Tightly Coupled Instruction Memory
Tightly Coupled Instruction Memory
Instruction Bus
Data Bus
Tightly Coupled Data Memory
Tightly Coupled Data Memory

Figure 2.2: Simplified block diagram of Nios II soft-core processor

and the instructions paths are independent. The most used version was the *Fast* one, which also occupies more internal logic. Furthermore, it supports up to 8 masters ports (4 for instructions and 4 for data) that can be used to directly connect to on-chip memory and form tightly coupled memories. The simplified block diagram is depicted in 2.2, where the main components inside and the most important signals and ports are shown.

Another great advantage of this CPU is the feature of letting the user to add more instructions to the arithmetic logic unit (ALU) in a hardware level. These custom instructions are handled by Nios II exactly in the same manner as all others, and there is only the need to respect the signals and timings.

2.2.2 Custom Instructions

One of the greatest advantages of using soft-core processors like Nios II is the possibility of extending his usability by adding custom instructions to the processor instruction set. In this way, it is possible to accelerate time-critical software algorithms by reducing a complex sequence of standard instructions to a single instruction implemented as hardware. In our system, the usage of a custom instruction allowed a linear feedback shift register, originally built on software, to increase performance from several thousands of clock cycles to only two, as it will be described later on chapter 4. The custom instruction logic connects directly to the Nios II ALU as shown in figure 2.3.

As explained in [17], Nios II processor supports up to 256 different custom instructions and each one of them can have different configurations. In this thesis a multi clock cycle custom logic block of fixed duration is used, but it is possible to have only a combinational type, an extended type, a type to access internal registers or one that works as an external interface. The simpler case is the combinational custom instruction, which consists of a logic block that is able to be

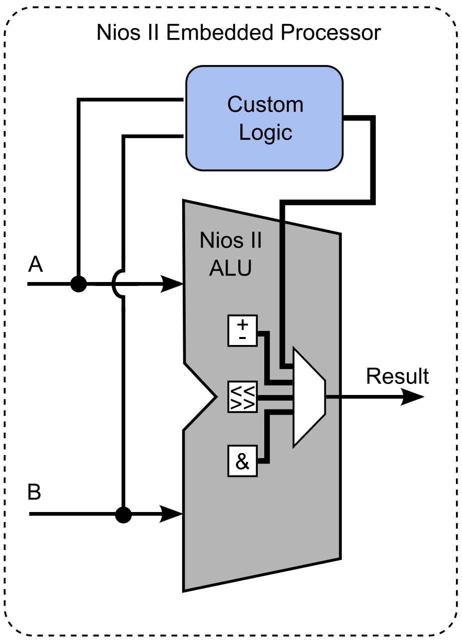


Figure 2.3: Custom instruction logic connects to the Nios II ALU, based on the picture from [17]

completed in a single clock cycle. However, when there is a big delay between the input and the output, the maximum frequency (f_{max}) of the CPU can be highly decreased. In this situation is better to separate the combinational circuitry into several stages, and to use a multi-cycle custom instruction instead.

The extended custom instruction uses another input signal, $n[7..0]$, that can be used to multiplex several different instructions (up to 256 since it is 8-bit wide) inside the same custom instruction module. The internal register file custom instruction allows logic to access its own internal register file that can be either from the Nios II processor's register file or from the custom instruction's own internal register file. Finally, the custom instruction used as an external interface allows the designer to communicate with logic outside the processor's data path.

The multi-cycle custom instruction only requires the following ports; *clk*, *clk_en* and *reset*. All of the others are optional and they only depend on the application.

In the timing diagram of the figure 2.4 it is possible to see its functionality. Processor asserts the active high *start* port on the first clock cycle of the custom instruction execution. At this time, the *dataa* and *datab* ports, if they are used, have valid values that remain valid throughout the duration. In the case of having a fixed length, the processor waits a specified number of clock cycles, and then reads the *result*. The number of clock cycles that it has to wait is specified during system generation. If the length is variable, the processor waits until the active high *done* is asserted and reads the *result*. Of course, it can be further optimized by combining several different types of configurations in the *same* custom instruction.

After the hardware is finished, there is the need to use it through the software. To simplify the procedure, the Nios II IDE generates macros that allow easy access from application code

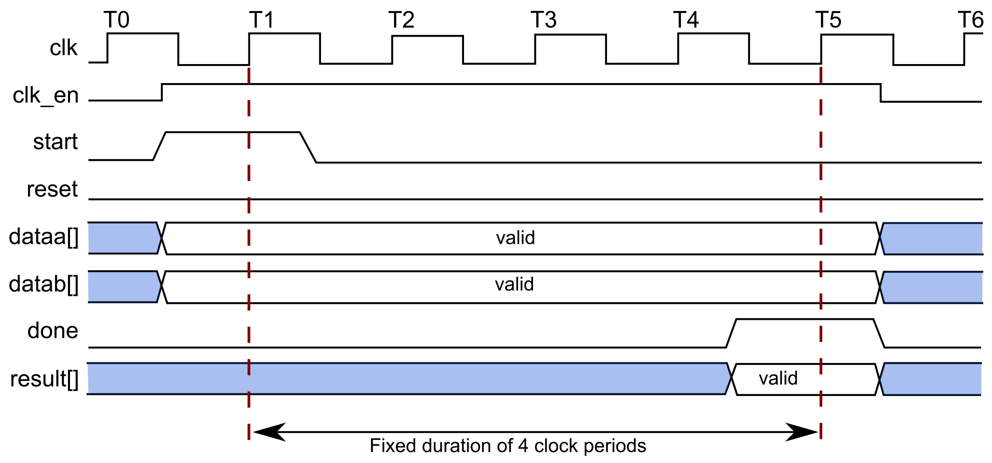


Figure 2.4: Timing diagram of a multi-cycle custom instruction, from [17]

to custom instructions. Those are specified in the **system.h** header file, where all the system information from that processor *point of view* is described.

2.2.3 Avalon Switch Fabric

As described in [18], Avalon Switch Fabric is the interconnect interface designed by Altera to connect all the components inside the FPGA. Usually, in the typical bus there is only one master access at a time. However, this switch fabric lets multiple masters to operate simultaneously due to a slave-side arbitration scheme. In the figure 2.5 is shown a basic multiprocessor platform with the arbiter represented in the slave-side. This is the default topology of the fabric, and is the ideal interconnection mechanism for embedded systems. It uses a partial crossbar switch which is a mixture of the traditional bus and the full crossbar switch. The normal bus does not allow to have a big concurrency because there is only one master at a time, but it can operate at higher frequencies. Meanwhile, the full crossbar switch is meant for parallel activities since there is a dedicate path between the master and the slave, but it grows exponentially as more masters and/or slaves are added.

Another important advantage is that it is extremely easy to set-up all the connections using SOPC Builder [19].

There are six different available interfaces. The most common ones, in the optical of the user, are the Avalon-MM (Memory Mapped Interface) and Avalon-ST (Streaming Interface). The first is the typical interface for master-slave connections and the second one is very useful to stream data between different hardware components. There is also a tristate interface to handle multiple peripherals that can share data and address buses. In this way it is possible to reduce the pin count of the FPGA and the number of traces on the PCB (printed circuit board). Besides those, there are three more specific interfaces: Avalon Clock, Avalon Interrupt and Avalon Conduit. The Avalon Clock is responsible to drive and receive the clock and reset signals in order to synchronize all interfaces. The Avalon Interrupt allows components to signal events to other components.

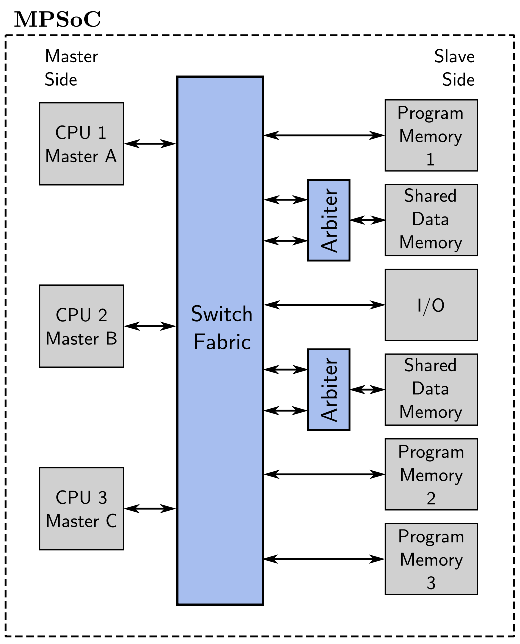


Figure 2.5: Partial crossbar switch

Finally, the Avalon Conduit that allows signals to be exported out so they can be connected to other modules of the design or FPGA pins.

Despite of the fact that it is relatively easy to build a SoC in SOPC Builder, several optimizations are required to take the best of the Stratix II FPGA chip, where some of them are available in [20].

2.2.4 Performance Counter

This hardware module from Altera allows to check the time of a given computation inside a Nios II soft processor. It is very important to accurately check the WCET (worst case execution time) of each process (or task), and to evaluate the generated overhead by all synchronization mechanisms and by the μC/OS-II, which is the embedded operating system used for control and processing the retrieved data.

The main benefit of using the performance counter is the accuracy of the results due to the fact that it is unobtrusive. It does not use any RAM and requires only a single instruction to start and to stop up to 7 different segments of code. However, it costs several FPGA's logic elements (LEs) so it is not recommended to use it when there is not the need to verify the timings with such precision.

It has a slave interface to the Avalon Switch Fabric and, in a multi-processor system, there should be one performance counter per processor connected to the corresponding data master interface. Next, the macros used to control the measurements are shown:

```
//Base Address of performance counter defined in system.h header
#define PERFORMANCE_COUNTER_1_BASE 0x60c0

PERF_RESET (PERFORMANCE_COUNTER_1_BASE);
//Macro to start running the global counter
PERF_START_MEASURING (PERFORMANCE_COUNTER_1_BASE);
//Macro to start measuring the segment 1
PERF_BEGIN (PERFORMANCE_COUNTER_1_BASE, 1);

//code to be measured is placed here.

//Macro to stop measuring
PERF_END (PERFORMANCE_COUNTER_1_BASE, 1);
//Macro to get the number of cycles of the measurement
time1=perf_get_section_time(PERFORMANCE_COUNTER_1_BASE,1);
//Macro to stop running the global counter
PERF_STOP_MEASURING (PERFORMANCE_COUNTER_1_BASE);
```

The performance counter module has 64-bit counters that must be connected to the same clock of the measured CPU. In our systems the clock frequency is set to 50 MHz. Due to the fact that the software variable to store the number of cycles is only 32-bit wide, it was easy to overflow. Thus, it is only possible to count the number of clock cycles up to around 85 seconds. However, it is not changed because it is enough for our experiments.

2.2.5 Timer

A timer is a basic requirement for an embedded system in order to work properly. Specially for real-time applications, a real-time clock (RTC) is mandatory to keep track of the time. In this work, there is no need to keep track of the time, but a timer is needed to generate periodic interruptions (1 ms) for the real-time operating system. In this way, it is possible to control the time that certain tasks might take by counting the number of IRQs, although this task is hidden inside the kernel. The timer used is described in Altera manual [21] and has 32-bit and 64-bit counters.

2.2.6 Mailbox

The Altera Mailbox is a hardware component that works as a FIFO, and lets the delivery of 32-bit messages from one component to other. It contains mutexes to ensure that only one processor modifies the mailbox contents at a time. It is used in conjunction with a separate shared memory, which is used for storing the messages. It is also described in the Altera manual [21].

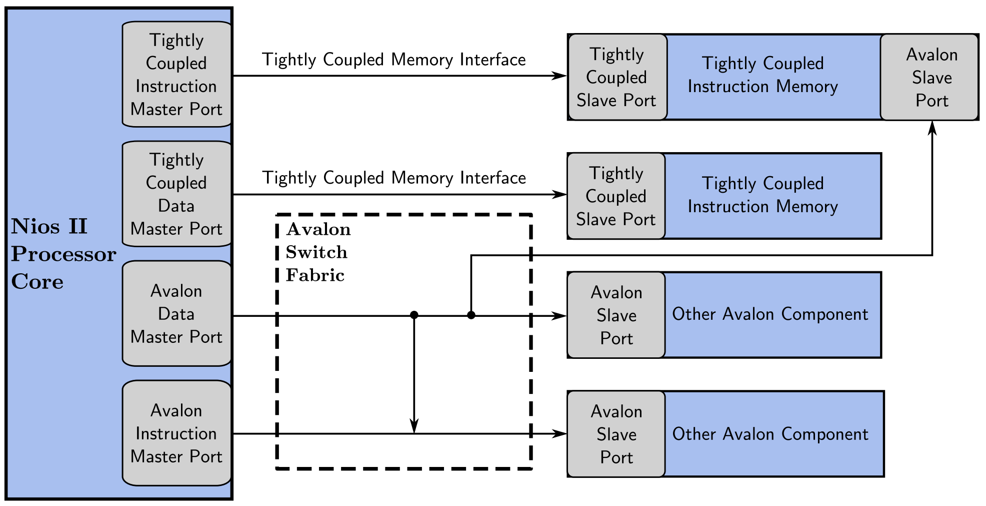

Nios II Processor Core
Tightly Coupled Instruction Master Port
Tightly Coupled Data Master Port
Avalon Data Master Port
Avalon Instruction Master Port
Tightly Coupled Memory Interface
Tightly Coupled Memory Interface
Avalon Switch Fabric
Tightly Coupled Slave Port
Tightly Coupled Instruction Memory
Avalon Slave Port
Tightly Coupled Slave Port
Tightly Coupled Instruction Memory
Avalon Slave Port
Other Avalon Component
Avalon Slave Port
Other Avalon Component

Figure 2.6: Block diagram of a simple Nios II system with TCM showing the Avalon interfaces

2.2.7 Tightly Coupled Memory

Nios II permits to add up to 8 special ports in order to easily allocate direct links to the on-chip memories, avoiding the normal rules of Avalon Switch Fabric. In this way, the bottleneck effect and the signaling overhead is highly decreased and the speed is almost the same as if it was accessing cache. However, it is not possible to use TCM as shared memory since only one master port can be connected to each memory.

There are two different purposes, as shown in figure 2.6. One requires a dual-port on-chip memory and only handles the instructions (or the *.text* segment of a typical program). The other type is for data storage, or all the other program segments.

A single processor test system with the fastest available components is included to check the performance at 50MHz clock speed according to the tutorial [22]. The connections between components are depicted in the figure 2.7 and specifications are the following:

- **CPU** — Fast version of the soft-core Altera Nios II with tightly coupled master interfaces for instructions and data;
- **TCM for Data** — Tightly Coupled Memory for data storage with 32 kbytes;
- **TCM for Instructions** — Tightly Coupled Memory for instructions storage with 64 kbytes;
- **Interface** — JTAG UART module that is the I/O interface connected to the USB Blaster for debugging and data communication with the host computer;
- **Timer** — 1ms timer to generate the interruptions needed by Nios II;
- **Shared Memory** — On-chip memory with 16 kbytes.

In the table 2.2 there is a comparison of performance achieved by three different choices of memory. Note that TCM are on-chip memories with a dedicated link. Those values are the result

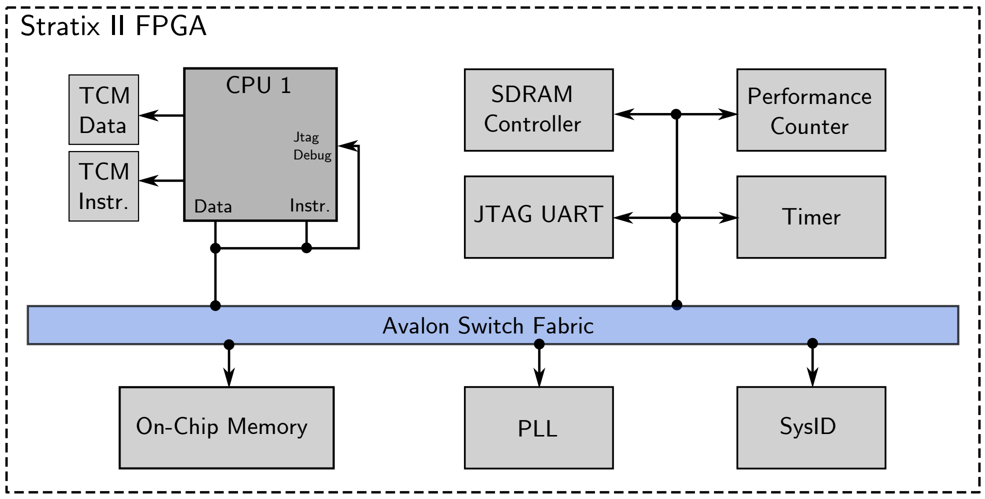


Figure 2.7: Block diagram of a simple Nios II system

of writing and then reading a block of 320 32-bit words (i.e. 10,240 bits or 1.25 kbytes of raw data). In this test, the times are already optimized due to the fact that the access to the memory is sequential, without the need to give the bus access to another master or to repeat again the start-up procedure to initiate communication. The Avalon Switch Fabric works better in this way; otherwise, the times for the on-chip memory and for the SDRAM would have been worse. This issue will be debated again with more depth in chapter 3.

Table 2.2: Comparison between the performance of three different types of available memory on Stratix II FPGA

	Time	**Clock Cycles**
Tightly Coupled Memory	90.02 μs	4,501
On-Chip Memory	119.54 μs	5,977
On-Chip Memory (cached)	94.88 μs	4,744
SDRAM Memory	162.00 μs	8,100
SDRAM Memory (cached)	89.90 μs	4,495

The purpose of this performance test is to show that it is possible to use tightly coupled memories for the worker's CPUs so they can achieve considerable better performance. Using this kind of memory, the system works faster and simplifies the architecture because the bottleneck effect is completely avoided since there is a direct connection to the CPU.

2.2.8 IRQ Mailbox Core

This hardware module is responsible to generate an interruption with the corresponding description and was designed by Hang Zhang in VHDL. It is very useful to synchronize a CPU, running an operating system like μC/OS-II, with some event that occurred in some other component. It is used to generate a truly random seed in the random generator; however, it might be used for

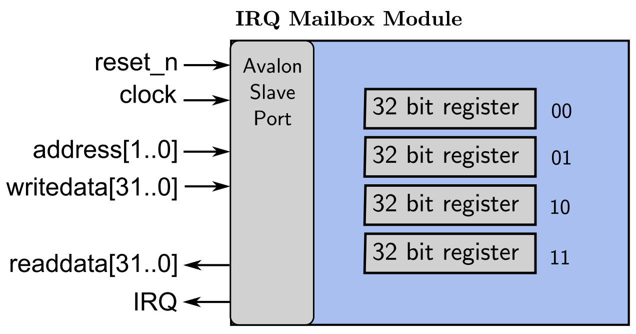


Figure 2.8: Block diagram of the IRQ mailbox module

another applications like triggering tasks for a different processor. For instance, in a streaming application where tasks are mapped on several different processors and there is a scheduler running on another dedicated processor, this module allows the scheduler to know when some particular job is done so it can trigger another job in the same processor.

Figure 2.8 shows the block diagram with the corresponding I/O ports required to work with the Avalon Switch Fabric. Basically, it has four 32-bit registers and, when one of them is written by software, the core generates an IRQ alerting the existence of a new message.

2.3 MPSoC Platform

In this section there is an overview of all different platforms built with the components previously described in this chapter. The goal is to explore different approaches for the communication inside MPSoCs, using the same interconnection interface.

2.3.1 Introduction

A typical MPSoC consists on multiple CPUs on a SoC, aiming to have a parallel application to take full advantage of the hardware [23]. Traditional Symmetric Multiprocessing (SMP) is a useful solution where the performance of a certain application is scaled up by adding more similar processors. However, by having a heterogeneous system like for instance a CPU and a GPU inside the same chip, new challenges arise and they require new methods to be designed in order to greatly improve overall performance. As explained in [1], heterogeneity presents many problems during their design. In this thesis some of those problems are minimized since that all the CPUs are Nios II with similar instruction sets and the synchronization mechanisms are mutual. Furthermore, all the custom hardware follows the Avalon Switch Fabric rules. However, the problems that still remain are the following, but they will be discussed in the next chapter since they belong to a higher level:

1. How is it possible to evaluate the load balance of the processors in a heterogeneous MPSoC. There is also the challenge of application partitioning/mapping on independent processors to ensure dependent services.

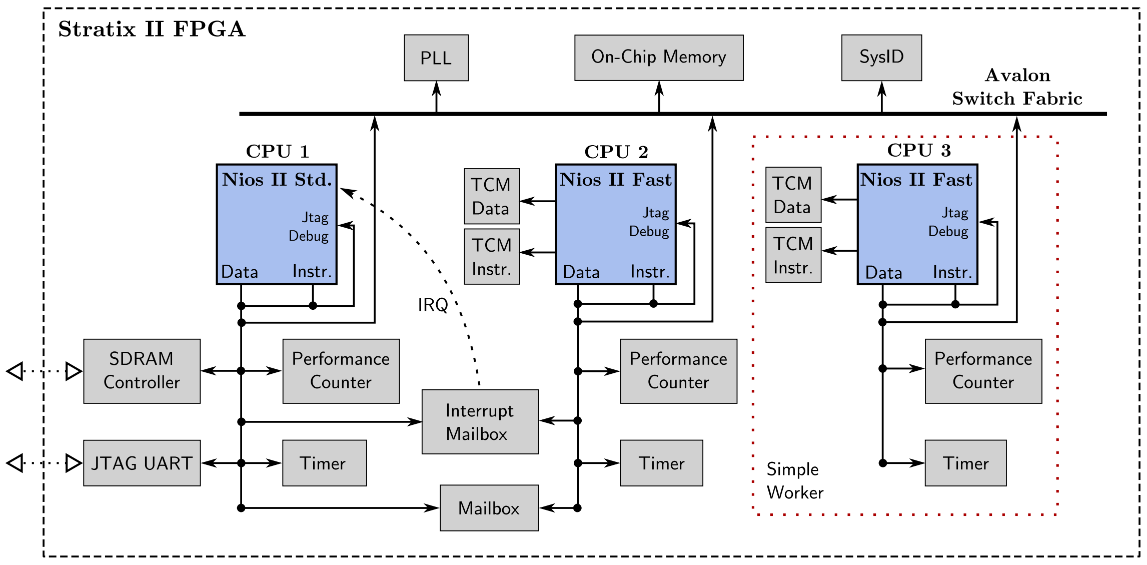


Figure 2.9: Hardware architecture with 3 Nios II processors

2. Trade-offs between performance, power and real-time programming are very difficult to achieve and its very application dependent.

2.3.2 Architecture

A simplified architecture of a MPSoC was already presented in the figure 2.5, where the arbitration modules are needed if there is a concurrent access by two masters to a slave. Using the SOPC Builder, the work of connecting all the modules is dramatically decreased since it allows to use directly the interconnect fabric provided by Altera. In fact, the user does not need to be concerned about the arbitration *inside* the fabric. During the generation of the system, the SOPC Builder creates a description of all system in a hardware description language (HDL) that is later compiled by Quartus. Thanks to these tools, it is possible to add as many processors and peripherals to a system as desired. The design challenge in building multiprocessor systems now lies in writing the software for those processors so they can operate efficiently together, thus avoiding conflicts.

The most used platform in our applications is depicted in the figure 2.9. It has 3 Nios II processors, where two of them are the fastest version with tightly coupled memories for data and instructions. The third one is a standard version that acts as both the scheduler and the interface to the outside world.

The latter is connected to the SDRAM controller that communicates to the external SDRAM chip, where it stores the data and instructions, and to the JTAG UART module, which in turn is connected to the USB Blaster peripheral that is used to program the FPGA and to send the values to the user console in the computer. All the processors have a JTAG port for debugging that allows the IDE tools to run the code step-by-step and to set break-points.

All processors have a private performance counter and a timer connected to the corresponding data master port, and they all have access to the on-chip memory (8 kbytes of size). There is also a global System ID module very useful to check if a compiled software belongs to the system it was created for, and a Phase-Locked Loop (PLL) module that is a control system meant to keep the phase of the clock always synchronized. In fact, it has a -3ps time shift in order to avoid some problems related to the timing loss by the SDRAM controller.

These are the main modules used since other aspects were changed according to the target application, specially the amount of space available on the memories used as TCM and the mailboxes. However, in the picture is shown the most used architecture during this thesis.

In the smaller square there is a simple worker that can be replicated as many times we want, as long as there is enough logic (and on-chip memory) space inside the FPGA. For the first application it had also the Interrupt Mailbox and a Mailbox connected to the CPU 3. For simplicity it was shown only for CPU 2. The Interrupt Mailbox and the Mailbox needs to be connected to the corresponding data master ports, since they can be seen by both processors. The interruptions generated by the Interrupt Mailbox also uses the same interconnection fabric but is represented with a dashed arrow to better explain the concept.

Table 2.3: Summary of FPGA utilization

	Used	**Available**	**Used(%)**
ALUTs	8,402	48,352	17%
Dedicated Logic Registers	6,245	48,352	13%
Estimated ALMs	5,586	24,176	23%
Total LABs	828	3,022	27%
I/O Pins	67	492	14%

The table 2.3 shows the FPGA utilization of the platform. The area is reported in terms of ALUTs and these statistics have been obtained by synthesizing the platform with Quartus 10.0 into Stratix II EP2S60. The first experiments were made with Quartus 7.2 which was, back then, more stable and reliable before the release of 10.0. The overall logic utilization was only 23%, hence we could perfectly add more components thus increasing the complexity according to the application. The estimated number of adaptive logic modules (ALMs) and the number of total logic array blocks (LABs) reflects partially or completely used units since that they are adaptive.

On the other hand, as said in the beginning of this chapter when we introduced the FPGA board, a limitation is the amount of on-chip memory available. In fact, we could not use the two M-RAM blocks available in order to avoid eventual problems. As shown in the table 2.4, the most used memory block was the true dual-port M4Ks. Despite the fact that only 34% of the total on-chip memory was used, in fact this value is 64% when not considering the M-RAM blocks.

Table 2.4: Memory Usage

	Used	**Total**	**Used(%)**
Block Memory Bits	865,792	2,544,192	34%
M512s	6	329	2%
M4Ks	219	255	86%

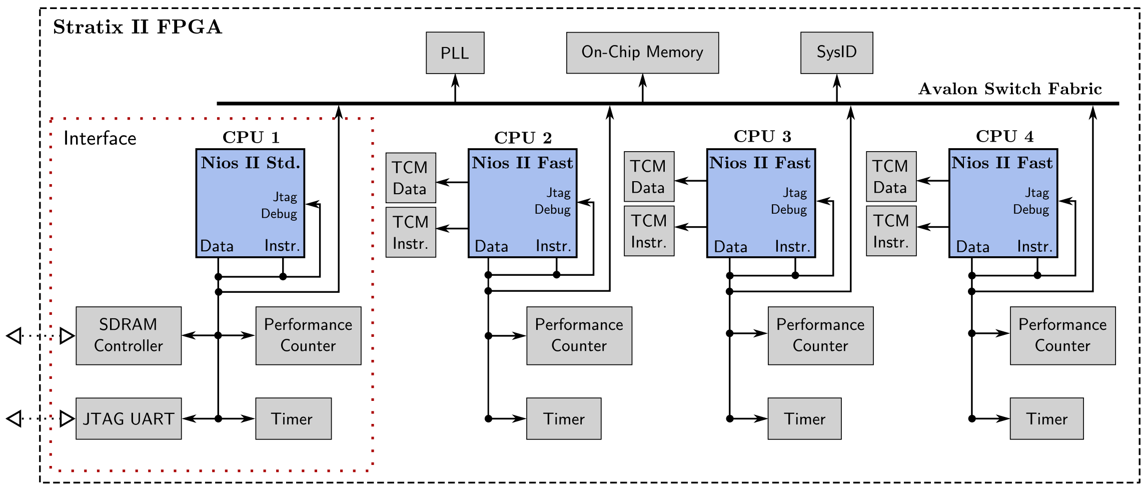


Figure 2.10: Architecture of the MPSoC with 4 CPUs

In chapter 4, in order to increase performance and have a totally on-chip processing, we added one more processor that was used as the scheduler. This platform is depicted in figure 2.10 and the mailboxes were removed. Thus, all the communication between processors were made using FIFOs stored in the on-chip shared memory.

The summary of the FPGA utilization is in the table 2.5 and it is possible to see a bigger occupation. The overall logic occupation has increased from 23% to 30%. The CPU 1 is now responsible to give the start/stop order to the applications and to print out the results that have been written on the shared memory by the other processors. Once again, the real amount of on-chip memory used is 72% when not considering the M-RAM blocks.

Table 2.5: Summary of FPGA utilization for the 4-core MPSoC

	Used	**Available**	**Used(%)**
ALUTs	10,508	48,352	22%
Dedicated Logic Registers	8,409	48,352	17%
Estimated ALMs	7,127	24,176	29%
Total LABs	1,064	3,022	35%
I/O Pins	67	492	14%
Total Block Memory Bits	985,152	2,544,192	39%

2.4 Software

In this section the software components used are exposed and debated.

2.4.1 Introduction

After having the hardware platforms explained, it is important to describe the middle and higher levels parts (i.e. the hardware abstraction layer (HAL) and the software).

In order to run the software on a specific system, an abstract layer that describes all the devices is needed and it is very important since it allows, for instance, to run the same operating system on different processors. Its function is to interact directly to the hardware instead of leaving that tedious task to the programmer. Therefore, the HAL requires less processing time than application programming interface. In Nios II systems, the HAL is a lightweight runtime environment that provides a simple device driver interface for programs to connect to the underlying hardware. It is integrated to the standard ANSI C library in a way that the user can use familiar C functions, such as `printf()`, `fopen()`, `fwrite()`, etc.

A typical Nios II program is divided in several segments as shown in the table 2.6.

Table 2.6: Division of a typical Nios II program

.heap	place where the fixed variables are stored (i.e. declared variables, global variables, static local variables, etc)
.stack	place where the dynamic variables are stored (i.e. local variables and function parameters)
.bss	non-initialized variables
.rodata	read-only data
.rwdata	read and write data
.text	code segment

To learn and verify the available optimizations during the compilation, a simple program (the one used as an example in the Altera's tutorial about the tightly coupled memories, available through [22]) was customized to fit in the test system exposed in the section 2.2.7, regarding the usability of TCM in our applications.

Therefore, without the use of any function of **stdio.h** library, with best possible optimization (-O3), and with the following flags enabled ("enable reduced device drivers" and "enable C small library"), we have experienced a reduction of 62,5% for initialized data and 42% for program size. The reason to not take into account the **stdio.h** library is because it will not be used during the normal working mode since that there is no I/O needs, only pure raw calculations. It is obvious that this reduction is completely application dependent. We just wanted to show what can be used to optimize the compilation of the C programs into machine language.

Table 2.7: Optimization achieved in a simple piece of code, using the typical *Gcc* compiler optimization directives

	Program Size	**Initial Size of Data**
Before	12KB	8KB
After	7KB	3KB

2.4.2 μC/OS-II

This is a real-time operating system designed for embedded applications written by Jean J. Labrosse and fully described in his book [24]. μC/OS-II is currently maintained by Micrium Inc. [25], and can be used as a hard real-time RTOS where *all* the dead-lines and constraints must be respected, otherwise it can result in fatalities. In fact, according to the book, in July of 2000, after extensive tests, μC/OS-II was certified to be used in avionics of commercial aircraft by the Federal Aviation Administration (FAA), which means that the software is both robust and safe. In this section there will be only a brief review of its operation with a special focus on the functionalities used in this thesis.

The main benefits of μC/OS-II is that it is simple to use, to implement and very effective. It has been ported to more than 100 microprocessors and micro-controllers, including Nios II. Altera allows an easy way to use it through their integrated development environment (IDE). It is a very scalable, preemptive real-time, deterministic and multitasking kernel that can manage up to 56 user tasks. In fact, it can manage up to 64 tasks but the four highest and the four lowest priority tasks are reserved for its own use. The lower the value of the priority, the higher the priority of the task. This value is also used as the task identifier.

The scheduler uses a special type of Rate Monotonic Scheduling (RMS), where the tasks with the highest rate of execution are given the highest priority. However, in this case the priorities are given by the user, so the worst case execution time (WCET) must be known.

In RMS there are three basic assumptions:

- All tasks are periodic;
- They do not synchronize with one another, share resources, etc;
- Preemptive scheduling is used which means that runs the highest priority task that is ready.

Under these assumptions, let n be the number of tasks, E_i be the execution time of task i, and T_i be the period of task i. Then, all deadlines will be met if the following inequality is satisfied:

$$\sum E_i/T_i \quad \leq \quad n\left(2^{1/n}-1\right). \tag{2.1}$$

This is not good for our applications because two or more tasks should have similar priorities since we are trying to speed-up the system in order to analyse the synchronization and the communication between tasks. However, when migrating the framework for a real-time scenario, we need to ensure that the tasks only executes when we want. In the topology evaluated in the next chapter, if both *Source* and *Destination* tasks are meant to run inside the same processor, it is convenient that *Source* has higher priority, otherwise the *Destination* will never get the results. On other words, in a chain of events, if the first one does not work, all the others will be stalled.

The biggest advantage of using this RTOS when we do not want in fact, the real-time capabilities, is due to the fact that it is easier to process interruptions, and to verify the CPU usage

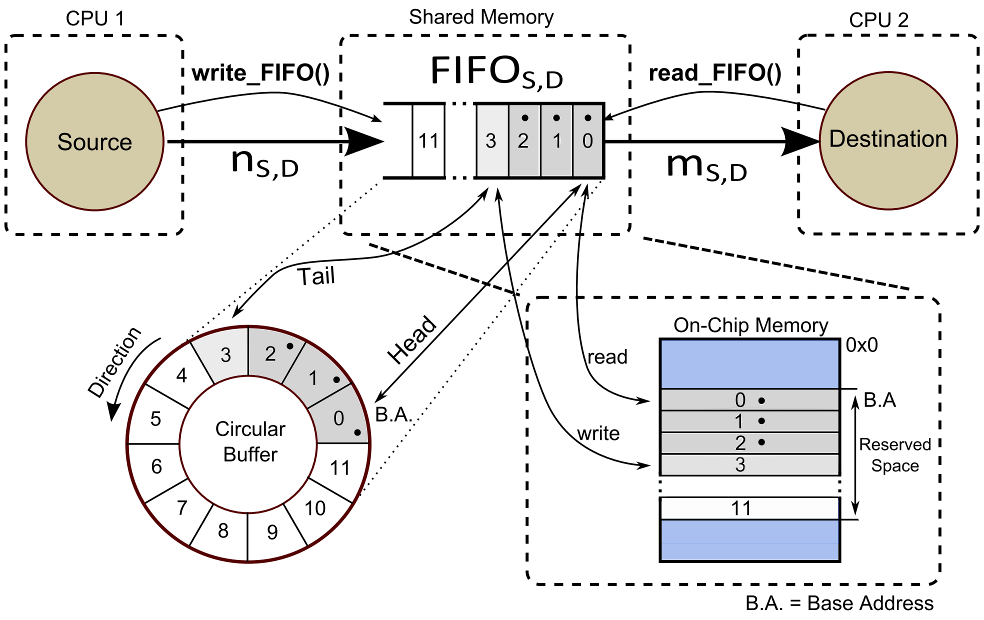


Figure 2.11: Communication between two processors using a FIFO stored in the shared memory

rate. Furthermore, we pulled it to the limits in order to see how well it can handle inter-tasks synchronization using semaphores, delays, priority switching when compared to a single task, that can perfectly run without any operating system.

2.4.3 FIFO

The controlling mechanisms of the buffers used for inter-processor communication are made in software, so they are not highly optimized for performance. However, they are simpler and can be extended in a future work as hardware using custom instructions for example. All the values are stored inside on-chip memory.

Many functions were made in order to allow non-blocking access to the shared memory, otherwise it would dramatically decrease performance. To make this working, different approaches were tried so when one processor is writing, the reader can not access to that memory slot. Due to the slave-side arbitration of Avalon Switch Fabric, the problem of having two masters trying to access the same memory location at same time is solved. However, we tried to improve performance since read/write operations are relatively slow.

In the figure 2.11 is shown the way of how it is possible to use software to send values (i.e. data tokens) to another processor. In the CPU 1 there is the *Source* task and in the CPU 2 the *Destination* Task. The read and write operations are non-blocking which means that, if for some reason they can not access, they will not be blocked waiting. That can occur if the sender tries to write in a full buffer, if the receiver tries to read from an empty buffer, or even if during the access to the FIFO another CPU is already taking control of the bus. In this last case, the Slave

Side Arbitration of the Avalon Switch Fabric does not give access in order to avoid the corruption of data.

That is why the processes need to keep checking to grant access since no static scheduling policy is being used. The structure of the FIFO is circular and it has two pointers. The *Head* points to the next value that is going to be read and the *Tail* points to the slot where a new value will be written. A circular way to organize the slots inside memory has advantages since when the pointers reach the limit of the reserved space, they jump back to the base address. This way the code is simplified since both pointers increment in the same direction.

In order to make the distinction between the empty and full states, a slot is *always* kept open. Therefore, the buffer depicted can use only 11 slots when in fact it has 12.

The FIFO is defined as a data structure. The type `alt_u32` is a way to say to the compiler that it is an unsigned 32-bit variable:

```
struct fifo{
  alt_u32 size;
  alt_u32 *baseAddress;
  alt_u32 head;
  alt_u32 tail;
};
```

The `write_fifo()` copies the data to the current position of the tail and then increments the pointer:

```
alt_u32 write_fifo(struct fifo* ff, alt_u32 data) {
  if(isFull_fifo(ff))
    return 1;
  else {
    *(ff->baseAddress + ff->tail) = data;
    ff->tail = (ff->tail + 1) % ff->size;
    return 0;
  }
}
```

The `read_fifo()` retrieves the value pointed by the head and then increments the pointer:

```
alt_u32 read_fifo(struct fifo* ff) {
  alt_u32 temp = *(ff->baseAddress + ff->head);
  ff->head = (ff->head + 1) % ff->size;
  return temp;
}
```

The following function is very useful to check the number of tokens inside the buffer:

```
alt_u32 backlogSize_fifo(struct fifo* ff){
  alt_u32 tail = ff->tail, head = ff->head;
  if(tail >= head)
    return (tail - head);
  else
    return (ff->size + tail - head);
}
```

2.4.4 Pseudo-Random Generator

The software version to generate random values is based on a linear feedback shift register. The original piece of code was taken from a white noise generator available in [26] as a *SystemC* component.

SystemC is a way to have an event-driven simulation kernel in C++ and it is based in a set of C++ classes and macros.

The code was customized for C, the shift-register is 16 bits wide and the generation of the first value inside the shift-register based on the the value of the `seed` is:

```
initial_SR_Content() {
        for(i=15; i>=0; i--) {
          if(seed>=pow(2.,(double)i)) {
                  shiftreg[i]=1;
                  seed-=(long int)pow(2.,(double)i);
          }
          else shiftreg[i]=0;
        }
}
```

It is important that the first value is well chosen, otherwise it can not “stimulate” the feedback and all the output values will be full of zeros or ones.

The linear feedback shift register was translated to Verilog in chapter 3 based on the following C function:

```
int my_rand() {
  int i = 0;
  int zw=0;
  if(shiftreg[12]==shiftreg[13]==shiftreg[14]==shiftreg[15])
    zw=1;
  else zw=0;                          // computing feedback bit;
  for(i=15; i>0; i--) {
    shiftreg[i] = shiftreg[i-1];      // shifting;
  }
  shiftreg[0] = zw;                   // writing the feedback bit;
  float val = 0.;
  for(i=0; i<7; i++) {                // result only with 7 bits;
    if(shiftreg[2*i])
      val+=pow(2.,(double)i);         // extracting random number;
  }
  return((int)floor(val));            // average around 63 after
}                                     // 1000 iterations.
```

Where the code requires more time to finish is during the extraction of the random number. In fact, it has to multiply 2 raised the number of the shift-register index that has a logical one in order to transform a boolean array into a decimal value. The problem of this is that when there are more logical ones inside the shift register, the time will also exponentially increase. The code included here is already a big simplification, since in the original one, the shift register had 32 bits but it was too slow when running on the Nios II.

Chapter 3

Generic MPSoC Application

In this chapter there is an overview of all different platforms built with the components described earlier and using the MPSoC described in chapter 2. The goal is to explore different approaches in the communication between heterogeneous components.

3.1 Introduction

In parallel computation there is the need to synchronize different tasks, either if they are executing inside the same processor in a multi-tasking environment, or if they are running on several independent processors, where true parallelism can occur. Streaming applications running on a multiprocessor system can be modeled in a graph that consists of an array of nodes, where in each node is performed a task. If they are running on different processors, there is a *pipelined* computing and performance can be increased resulting in a higher throughput. Between the nodes, there are channels with FIFO buffers to cope with peaks of data being transmitted and they can be also used for synchronization purposes. In this thesis data can be *anything*, hence the usage of an abstract type called token. In fact, we are dealing with an abstract model of flow that can be applied to many different applications, as long as their algorithmic structures matches the data-flow model.

To extend even more this sequential array, we have been inspired by the *MapReduce* framework developed by Google as seen in figure 3.1. It has been applied on a large clusters of computers in order to speedup a given task that requires the processing of a huge amount of data. It is based on the functional mapping of smaller amounts of data into less powerful computers, and, when they finish the work, the results are sent to another computer whose job is to concatenate all the intermediate results into the final one.

When this idea is applied as a data-flow model into small embedded systems, it is possible to parallelize some of the stages to improve the performance and better distribute the work among the processors. The simplest case that has been followed throughout this thesis is the one depicted

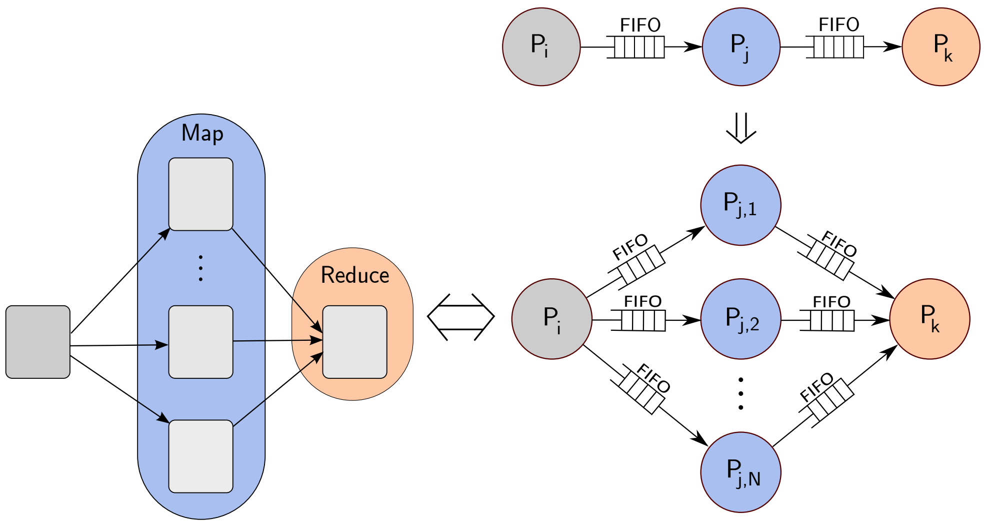


Figure 3.1: *MapReduce* framework translated to a data-flow model

in 3.2 with two workers. The task *Source* is responsible for the generation of tokens that are distributed by two workers, and the task *Destination* receives the results and organizes them. The goal is to analyze different approaches varying the size of the buffers, computational times and using different platforms formed by heterogeneous components.

3.1.1 Basic Concepts

This chapter is divided in three topics, where all the experiments were made using the MPSoC with 3 processors described in chapter 2. The workers are mapped on the two fast independent processors, and the *Source* and *Destination* tasks are mapped onto the processor that saves both data and instructions in the external SDRAM.

The first topic is based on the mapping stage of work among the processors, where a simple application was built in order to test the hardware platform and perform the preliminary experiments. The Mailbox + IRQ name arises from the fact that the scheduler fires the tasks by using the Altera Mailbox and, when they finish, interruptions are generated back to the scheduler running on the μC/OS-II so it can trigger new jobs.

The advantage of using the Mailbox is that Altera provides functions that enables a blocking read, which means that the task is stalled waiting for the order. This can be a good thing to reduce the power consumption while the system is idle. When the task finishes the current job, it signals the scheduler by generating an IRQ through the IRQ Mailbox. Later, the mailbox core was replaced by custom FIFOs in order to have a more flexible topology that can be extended according to the needs, specially in a future work where the FIFOs are connected to an external communication assistant.

Furthermore, we have included random times on the workers, with the purpose to verify if the

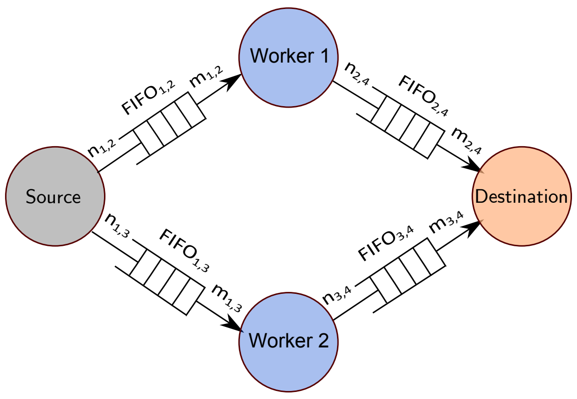


Figure 3.2: Simplest case of the MPSoC generic data-flow application

synchronization was working properly and in order to seek the biggest overhead caused by bottleneck. Although we started to customize the test application to generate random numbers, we only have used in a more flexible topology with only custom FIFOs between the processes. However, since the time between sending a message and receiving an interruption is always different, this can be used as a truly random seed to produce the first content inside the shift register used in the pseudo-random number generator.

The second topic is the natural extension of the first topology using interruptions to alert the scheduler that it can trigger a new job onto the specified worker. However, the Altera Mailbox was replaced by the custom FIFO so we could have a more generic and flexible platform. Also, the master CPU utilization rate was measured. Thus, it is possible to know if it can be used for other tasks without affecting the mapping and the reducing stages of the test application.

The third topic is focused on having only custom FIFOs between tasks. Here, the Mailbox is used again for the generation of the random seed, however the tasks only communicate via custom FIFOs and the workers do not produce any interruptions when they finish the current task.

In order to measure the theoretical times achieved by having parallel computation, we just need to add the reciprocals of each worker's time and take the reciprocal of the sum, according to the following equation:

$$EquivalentTime: \quad \frac{1}{T_{Equivalent}} = \frac{1}{t_1} + \frac{1}{t_2} + \ldots + \frac{1}{t_n}, \tag{3.1}$$

where n is the number of workers that are running in parallel and $t_1, t_2, \ldots, t_n$ are the respective times.

Hence, for two workers, $n = 2$, it is simplified to:

$$T_{Equivalent} = \frac{t_1 \cdot t_2}{t_1 + t_2}. \tag{3.2}$$

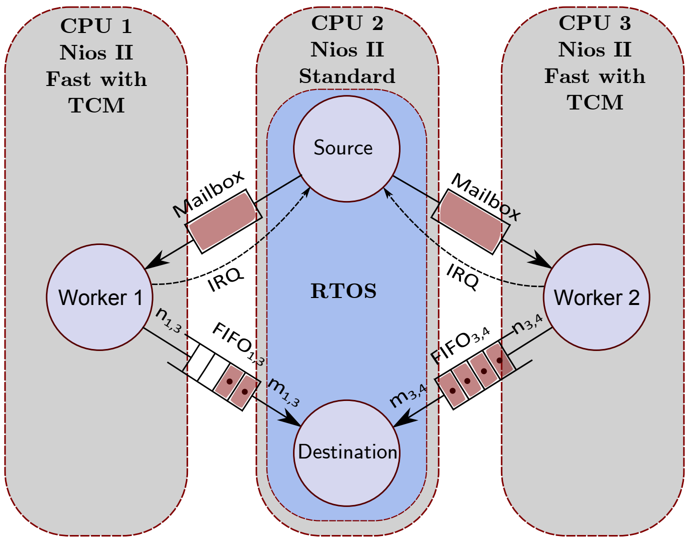


Figure 3.3: How the tasks are mapped onto 3 processors using the IRQ + Mailbox approach

This equation does not take into account the effect of the overhead caused by the communications and by the code that can not be parallelized, but it allows to discount the ideal time to the achieved result in order to find the amount of existing overhead.

Finally, one problem that has grabbed our attention was related to the Nios II processors cache. It was required to read data written by external modules and, in order to perform that, it was required to bypass all the variables that were shared. The cache is used to temporary store the most common data and it uses on-chip memory which is the fastest way to save data. It is a static memory which means that there is no need for periodic “refresh” (i.e. recharging the capacitors that holds the binary digits in SDRAM). In a two processor scenario where A and B perform read and write operations in the same memory space, if A changes a shared variable, B will not “see” because, according to his cache, it is the same as before. By doing a bypass of the cache, the problem of having one value that in fact does not exist is solved but with a lower performance. There are studies to use dynamic hybrid protocols to maintain the coherency among the caches of different processors so when one processor changes something on the shared memory the others' cache will not be corrupted [27]. However, this is beyond the scope of this thesis.

3.2 Mailbox + IRQ Synchronization

In this topology, described in the picture 3.3, the job information plus the synchronization are sent in bulk to the workers, as a 32-bit message, using the Altera Mailbox module. When the job is done, they generate an interruption back to the *Source* task and the results are sent through the FIFOs to the *Destination* task. By using interruptions its easier to guarantee the availability of the worker to produce more work without never completely use all the space of the Mailbox.

It is also possible to use this topology to make a more complex scheduler to fire the respective tasks in the due times. The hardware platform is the one depicted in 2.9, with three processors, in which the standard Nios II processor is simultaneously the simple dynamic scheduler and the interface through the JTAG UART module connected to the computer to review the results.

One simple application was created to check the performance using the idea of Google's *MapReduce*, where one pair (key, value) is sent and, in the end, all data is concatenated to produce the output result [12]. The goal is to retrieve in a text the number of occurrences of each letter as fast as possible, using two workers. This can be extended to search files and folders using a dedicated hardware module, even with more simple processors executing in parallel. Instead of using software searching algorithms it is possible to have a hardware component with a simple MPSoC designed to improve what is nowadays one of the major factor of performance limitation: the access to the stored data on slow memories.

3.2.1 Test Application

In embedded systems with multiprocessors there is the need to have separate programs running on each CPU, and all of them have different purposes, and uses different peripherals. The basic goal of this application is to test the overall mode of operation in order to evaluate if it is a good approach to increase the throughput of a data-flow graph by dividing specific segments, and mapping them onto different processors. Also, it is useful to test and learn the capabilities of the available technology.

However, while building this topology we have made a little more than just trigger tasks, wait some time and retrieve the results, such as the overheads of communication and the throughput. In fact, it was also used to test a PRNG (pseudo-number random generator) since we took the advantage of the fact that each processor is *always* executing a different instruction in the same instant of time.

The workers' job is only to count the number of occurrences of the letter received by the Altera Mailbox in a predefined text saved on memory. When they finish, the result is written in the FIFO so the master task can process a new letter. Also, they write a flag onto IRQ Mailbox module alerting that a specific job is done. This module then generates an interruption on the Master CPU so the *Source* task can trigger a new job on the available worker. This processor is running the μC/OS-II to simplify the handling of the interruption.

At this point, it was evident that the speed of processing each letter was too fast when compared to the mapping of the tasks, so a delay was introduced to mimic different workers speed. This delay was obtained through a while cycle of 10,000 iterations for worker 1 and 15,000 iterations for worker 2. The approximate time was in order of ms instead of μs, more precisely 3.23ms for worker 1 and 4.45ms for worker 2. The results are exposed on table 3.1.

Despite of the simplicity of this experiment, it was vital since it proved that the synchronization mechanism (both the IRQ and the buffers) was working properly and it was possible to know the overhead associated to the flow of information between the processors. Also, it proved that using multiprocessors in this kind of application increases performance as it was expected. Using both

Table 3.1: Results of the counting letters' application

	Average Time per Letter	**Total Time**	**Overhead**
Worker 1	3.42ms	88.82ms	0.19ms
Worker 2	4.64ms	120.79ms	0.19ms
Both Workers	1.98ms	51.60ms	0.11ms

workers showed a time reduction of 42% when compared to having only worker 1 and 57.4% with only worker 2. Furthermore, the overhead associated was about 0.19ms which is quite big if the workers are faster than that. Our initial thoughts were that it was caused mainly by the real-time operative system because it had to attend asynchronous interruptions. However, the RTOS scheduler had a small effect since that only one important tasks was running and the background ones almost were not called. Later we found that it was due to the slow accesses to the memory by the processor connected to the SDRAM, but this will be more discussed ahead.

3.2.2 Workers Random Delay

After having a multiprocessor application working, a random delay mechanism was included to see how the system handles different working times that in real non-deterministic systems exists. The WCET was fixed so it can be later applied to a real-time application.

Without the random generator the values were around 274.84ms for 10ms per letter, when the ideal value should closer to 260ms since there are 26 letters ($26x10 = 260$ms). Thus, the remaining 14.84ms, or 0.57ms per letter, was the overhead caused by the firing mechanism and the retrieval of the data through the FIFO.

Initially we used a software pseudo-random number generator (PRNG), described on the section 2.4.4 as a *SystemC* module, to produce a sequence of pseudo-random numbers from 0 to 127 (7 bits). Thus, the time of the workers was varying within the range from 10 to 20 ms. The biggest problem was that the multiplications required to generate a new value was too computational demanding. Furthermore, the time was completely unpredictable and never respecting the defined WCET.

The 7-bit version was already a simplification from the original algorithm of 16-bit, where initially the times obtained to reach the final letter, "Z", were from 642ms to 894ms (or about 233% worst) using only one worker with a theoretical fixed delay of 10 ms. The random values were generated in run-time every time a new task was fired, however they were not used since the goal was only to verify how worst the addition of the software PRNG could be.

With the 7-bit version the values were better, from 282.40ms to 336.65ms, but still not enough. The fact that the times are also random is not good for a real-time system except if the WCET is known. For a 64-bit shift register used in the random generator, the worst case execution time should be when all the values are binary 1 and there is the need to compute $2^0 + 2^1 + 2^2 + ... + 2^{63}$ to get the next random value. In software this proved to be too slow and that is why the need to build a random generator in hardware appeared.

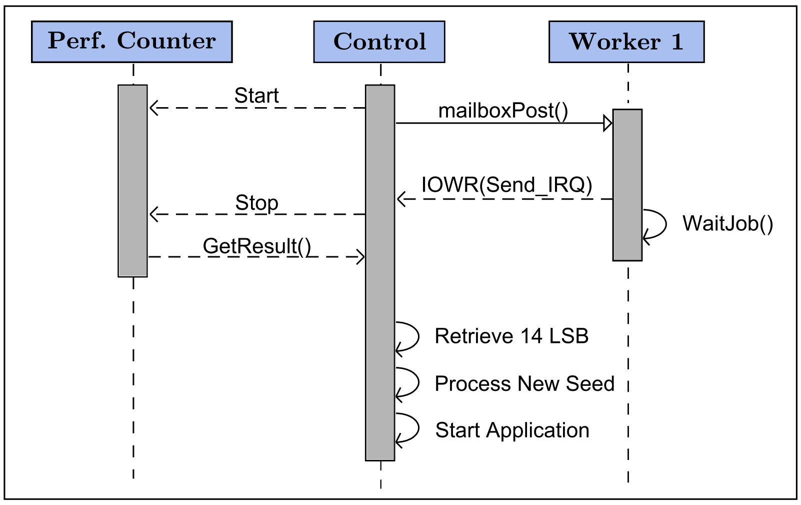


Figure 3.4: Sequence diagram explaining how the random seed is evaluated

However, before it was build, a way to overcome this flaw was to generate several pseudo-random values *before* the actual computation and save them in a table to this way check if the computation and synchronization was working. The times using a previously generated table were again about the same as before, like in the situation when no random generator was used; between 274ms and 276ms.

As explained before, the linear feedback shift register method to evaluate a sequence of pseudo-random values is *pseudo* due to the fact that it is deterministic. Knowing the seed, the method to obtain the first value inside the shift register, and the logic function used in the feedback, it is possible to know all the other values. To be more *random* and less *pseudo*, a method was evaluated to find a truly random seed.

Like is described in the figure 3.4, the seed is determined during the initialization of the program, when all the processors are already active and running. The cycles needed between the sending of a mailbox message asking for an interruption and the interruption are measured. The 14 LSB (less significant bits) of the number of cycles are *always* different so they can be used as a random seed. In the table 3.2 is shown two measurements that were used to get new random seeds.

Table 3.2: Example values to find a truly random seed

	No. of Cycles	**No. of Cycles in Binary**	**Time**	**Seed**
First run	506,634	111 10**11 1011 0000 1010**	10.132 ms	15,114
Second run	506,671	111 10**11 1011 0010 1111**	10.133 ms	15,151

After the seed is determined, the function `initial_SR_Content()` is called to compute the first content inside the shift register. The code is exposed in section 2.4.4 and, for the table above, the first three values of the sequence of random values are 103, 32, 78 for the first seed and 111, 38, 94 for the second one.

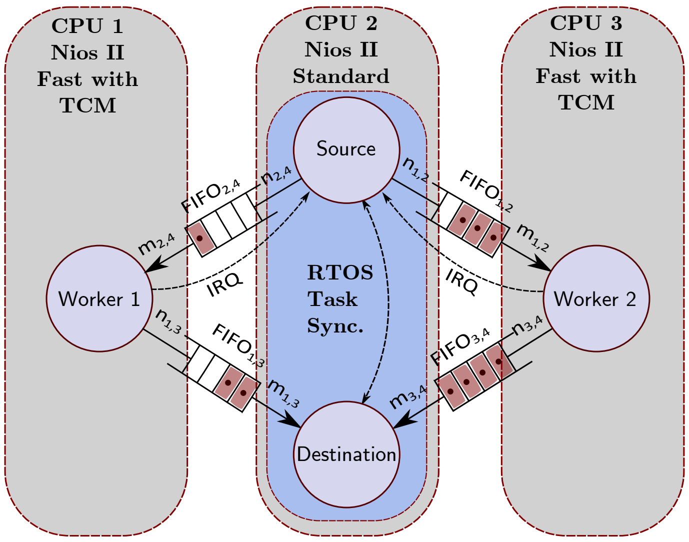


Figure 3.5: How the tasks are mapped onto 3 processors using the FIFO + IRQ synchronization

As said before, this is the result of having Nios II processors executing different instructions in a particular instant of time. It is also a direct consequence of the fact that all the components of this MPSoC are different (and processors start in a different instant of time). This is a special case of a mailbox plus interruption where the latency of the synchronization has been used.

3.3 FIFO + IRQ Synchronization

This section is a short extension of the previous one, regarding the synchronization using the Altera Mailbox, and interruptions that signals the conclusion of the task assigned to the respective worker.

The Altera Mailbox was replaced by the custom FIFO in order to have a more generic platform that was extended. A dummy application was made to check the rate of the master processor running the μC/OS-II operating system in the same conditions as before.

In order to evaluate the CPU utilization rate, only one task inside the operating system is created during the initialization of the operating system, `void TaskStart(void* pdata)`. In the beginning of its execution, the function `OSStatInit()` is called, and only after, the remaining tasks are created. This function is used to start the statistics task and define all the needed global variables. Every second this task runs in order to update the global variables used to check the current CPU usage, based on the amount of ticks that happened while the *Idle* task was running.

In the following table the results for a generic application are exposed, where the workers only have a fixed delay and, when they finish, the corresponding interruption is sent to the master CPU.

It is obvious that, when the theoretical time, given by the equation 3.2, is smaller, the master needs to execute more often, so the processor will have a bigger utilization rate. It was also interesting to verify that, under the same conditions than the experiment made using Mailbox

Table 3.3: Results of the generic application using FIFO+IRQ synchronization

Worker 1	**Worker 2**	**Result**	**Theoretical Result**	**Overhead**	**Master CPU Usage**
1.11ms	20.13ms	1.18ms	1.05ms	0.12ms	19%
3.23ms	4.45ms	1.94ms	1.87ms	0.07ms	12%
10.07ms	100.42ms	9.26ms	9.15ms	0.11ms	5%

(where Worker 1 and Worker 2 had a fixed delay of 3.23 ms and 4.45 ms), the overhead was smaller using the custom FIFO.

Table 3.4: Comparison of times between using Altera Mailbox and Custom FIFOs

	Altera Mailbox	**Custom FIFO**
Time to Process 26 Tokens	51.60ms	50.44ms
Average Time per Token	1.98ms	1.94ms
Theoretical Time per Token	1.87ms	1.87ms
Overhead	0.11ms	0.07ms

3.4 FIFO Synchronization

The synchronization between tasks is one important aspect of parallel computation. They can be inside the same processor where the need of some kind of scheduling policy is mandatory to achieve a good *quasi*-parallel computation, or outside using different processors or dedicated hardware modules. In the case of having heterogeneous components it is even more critical since the way of handling information is variable according to the component, hence the need of having a reliable mechanism to synchronize everything.

In this topic we focus the attention to the synchronization through buffers of the type first-in-first-out and we start again from the graph depicted in 3.2. The workers nodes are “dummy” in the sense that they are not doing anything useful. They only call the following C function to burn some time:

```
void wait_time(int number_of_iterations)
{
   while(number_of_iterations > 0)
      number_of_iterations--;
}
```

As said before, one of the major drawback of this thesis was due to the usage of SDRAM that has limited the theoretical high performance of the FIFOs stored in on-chip memory. The times to access them were worse than expected, hence the need to add a bigger dummy delay on the workers in order to mimic a more demanding computational time.

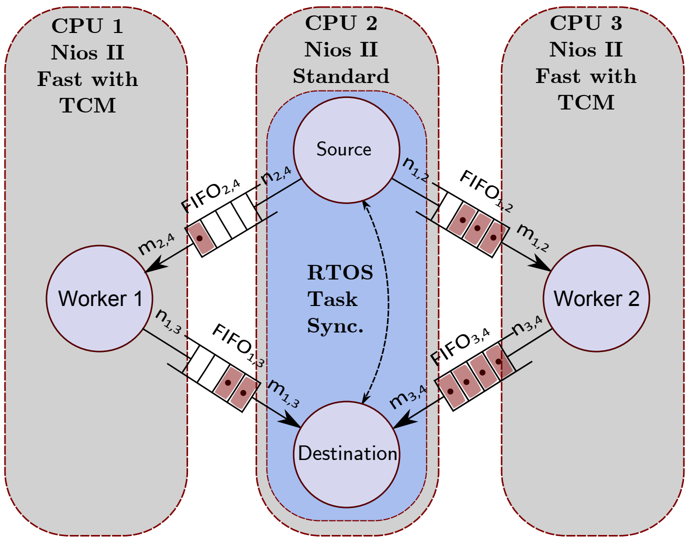


Figure 3.6: How the tasks are mapped onto 3 processors

One of the biggest challenges was to find a better way to synchronize the tasks on μC/OS-II. It was due to the fact that this RTOS is preemptive and all the tasks have different fixed priorities, which arises the problem of how to give the same priority to both *Source* and *Destination* task. Also, no interruption mechanism was used so all the processors were now working at full speed.

Despite that this topology is intended to be used in real-time applications, where usually the processing do not need to be the fastest possible as long as the deadlines and constraints are satisfied, by speeding up to the limits of the hardware it is possible to analyze and measure with more quality all the communication paths between tasks inside the operative system, and outside in different processors or peripherals.

Several tests were performed to find the best way. The mapping of the tasks to the processors was the one depicted in 3.6 where the CPU 2 is simultaneously the interface to the computer through JTAG UART module and it is responsible to generate and distribute the tokens (mapping/scheduling) in the *Source* task and to process the results (reduce) in the *Destination* task.

For a simple round-robin scheduling, one task is enough and it is the fastest way to guarantee the best time. It works by checking in a cyclic way the 4 abstract slots where each one of them represents the access to one of the 4 available buffers, like as if it was TDM (time-division multiplexing). The flow is depicted in the figure 3.7.

However, despite this is the simplest and fastest way to synchronize the system, if there are more parallel workers, the overhead will proportionally increase. Therefore, besides the dynamic scheduler introduced in the previous section that uses IRQs, different approaches were taken into account, and they share the following concepts: two independent RTOS tasks, one regarding the *mapping* of the tokens into the workers (*Source* Task) and another to *collect* all data produced by them (*Destination* Task). As previously said, this arises the problem that the priorities should be equal but this operative system does not allow that, so there is the need to manually force the

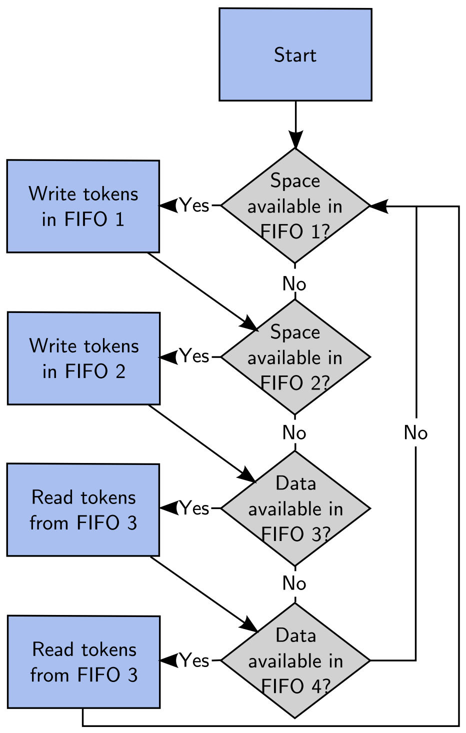


Figure 3.7: Flow diagram of the scheduler

RTOS to switch task as shown in the flow diagram 3.8. The *Mapping Task* has more priority than the *Reduce Task* since it has to run first to allow the workers to execute.

Three ways were tested; one using semaphores to pend the higher priority task letting this way the other with lower priority to run, another using the RTOS delay when the higher priority task finishes his current computation and needs to wait for more work, and finally, the third way was, during runtime, to manually switch the priorities forcing the RTOS scheduler to preempt to the lower priority task. This way it is possible to see the overhead generated by the context switching of the μC/OS-II.

Looking to the chart of figure 3.9 it is possible to see the different types and respective times of synchronization between tasks running inside the RTOS. The workers had random delays generated *before* the experiment took place, with a truly random seed, and the time frame is within the range from 10ms to 20ms. The bottom boundary was set by decrementing the value 20720 toward zero using a while loop and, for the upper boundary, the value was set to 41360. These values were discovered by a trial and error method by measuring the loop with the performance counter. Thus, since we have used the 7-bit software PRNG that generates numbers up to 127, the expression used to mimic a pseudo-random delay was $Var = 20720 + rand \cdot 162$, where Var is the amount that is decreased toward zero using the while loop, and $rand$ is the random value.

Since only 100 tokens were processed each time and we have made a lot of runnings, it was possible to know the most probable range by dividing the sum of all the random values by 100. Also, during all the experiments the buffers' size was fixed to 5 slots, except when using a μC/OS-

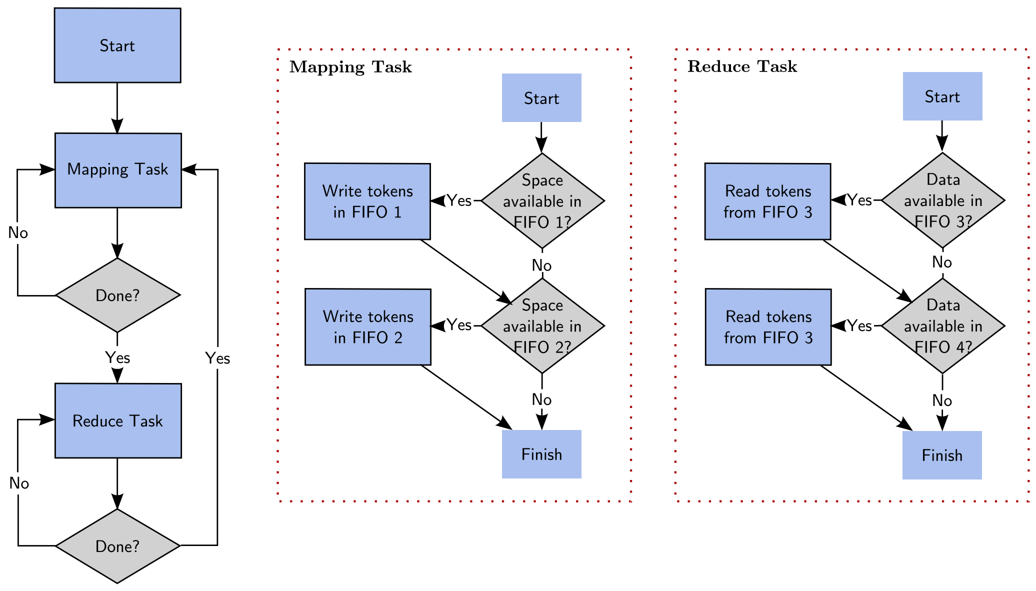


Figure 3.8: Flow diagram of the OS scheduler

II delay, `OSTimeDly()`, where we have increased the size to 20 slots. This function allows the calling task to delay itself for a user-specified number of periodic interrupts from the timer that is set to 1 ms.

The experimental results showed that if the buffers are too small, using delay to force task switching might increase the times due to the buffers overflow. Also, to change priorities in run-time obliges the operative system to execute a lot portion of background code for the context switching, resulting on a penalty of the overall performance.

Due to the fact that this requires a permanent check to see if work is available (i.e. if the buffers have space or have data to be processed), the control processor is always working at his full speed and this is not good for the energy consumption. In the ideal system it should be stalled most of the time to save power and only turned on to process or fire a new task. However, that requires an independent hardware module whose job is to check the current status of the FIFOs.

Therefore, we needed to have hardware buffers instead of software controlled ones, with the capability of generating interruptions. μC/OS-II have always a background task, called *Idle Task*, that reduces power due to the lower switching rate of the transistors - the main source of energy consumption in digital circuits. This task can even switch off the processor in battery operated devices. When this “supervision” feature of the buffers detects a new job to be handled, it could generate an IRQ in the operative system, thus resuming activity.

These experiments were possible because the computation time of the workers was bigger when compared to the control tasks. However, a huge overhead was detected and we had to find its origin. The delays on the workers were removed so they could work as fast as possible and the overall performance was measured. Both workers now only took 9.35μs to process one token and 1000 tokens were sent to them. This experiment took 288.64ms or 288.64μs per token

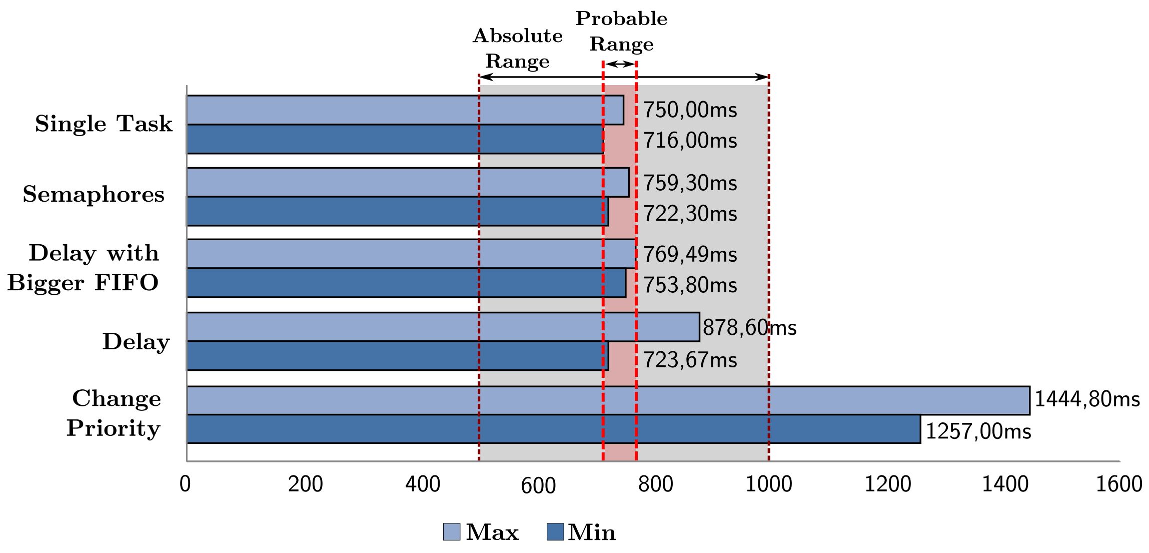


Figure 3.9: Chart with temporal results of 100 tokens processed by RTOS

which proved that the overhead due to the synchronization and communication was approximately 279.25μs.

Some reasons that could explain this issues are the following: the operative system and the fabric overhead were bigger than thought or, the most plausible one, the usage of the SDRAM by the control processor was not a wise choice since it is not fast. Furthermore, it is working at half of its recommended speed because it has to be synchronized with the clock of the processor. On the other hand, the on-chip memory used for data storage of the buffers is very fast (latency of 1 clock cycle to read), but the amount available on the FPGA chip requires a previous ponderation regarding its usage. However, in order to have a bottleneck effect on the fabric, a huge amount of data needs to be exchanged between the processors and peripherals. In fact, we could not notice any overhead caused by bottleneck during our experiments, possibly because we used too few masters competing for the access to the shared peripherals.

The single task version using operative system has been compared to a similar one without and, under the same conditions, the times were similar. In some cases, the RTOS version could be slightly faster.

Table 3.5: Performance of the on-chip memory FIFO buffers

	Nios II Fast + TCM		**Nios II Standard + SDRAM**	
	Time (μs)	**Clock Cycles**	**Time (μs)**	**Clock Cycles**
`write_fifo()`	3.42	171	15.00	750
`write_fifoN()` ($N = 2$)	9.88	494	45.04	2,252
`write_fifoN()` ($N = 4$)	16.90	845	72.76	3,638
`read_fifo()`	1.98	99	8.38	419
`read_fifoN()` ($N = 2$)	6.98	349	27.94	1,397
`read_fifoN()` ($N = 4$)	11.26	563	41.98	2,099
`backlogSize_fifo()`	1.24	62	4.90	245

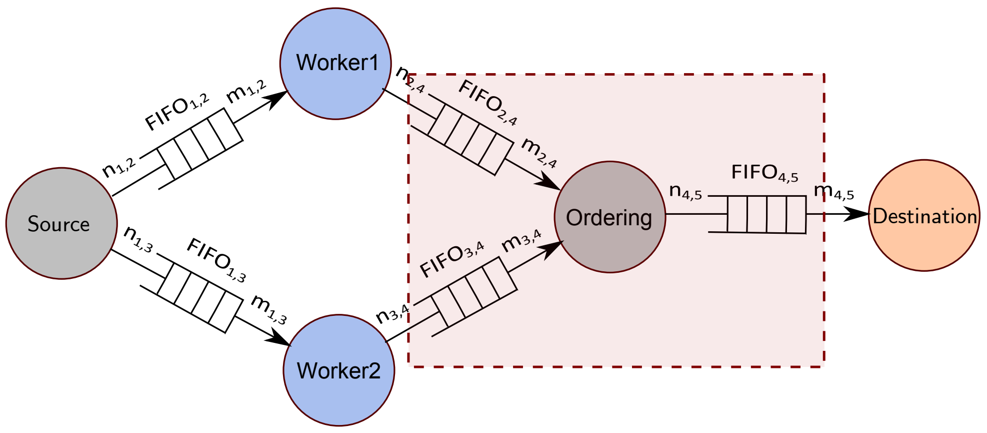


Figure 3.10: Ordering FIFO model

In the table 3.5, the measurements of the buffer accesses for the Nios II Fast with coupled memories and for the standard version using external SDRAM are presented. *N* means the number of tokens that are sent (or received) each time. The two functions, `write_fifoN()` and `read_fifoN()` have thread safe mechanisms in order to guarantee that no other processor accesses to the buffer while they are reading or writing. Both processors access to a shared on-chip memory but the tests were made separately to avoid any kind of bottleneck that could might occur. However, the performance differences between the two processors are quite significant. In fact, the time to assign a simple value to a variable (32-bit unsigned integer) only takes 5 clock cycles when using tightly coupled memories in contrast to the 18 clock cycles required by the Standard Nios II using the external SDRAM.

Later on chapter 4 we test ways to improve the performance using sequential accesses by sending a lot more tokens each time, and we also use a third Nios II processor with TCM to avoid the limitations of the available SDRAM.

3.4.1 Ordered FIFO

One problem that might happen in parallel computation is when there is the need of having some kind of order on the output. In the previous topics, the results of the workers depends only on the instant when they finish and they are not dependent among themselves. However, in many applications, like for instance in a NoC, the packets might travel through different paths to avoid bottleneck and there is the need to re-organize them in the destiny.

In this section, the goal was not to build a MIFO (multiple-input-first-output), but a way to retrieve and process indexes and, when they are not sequential, wait for the next one of the sequence. In the figure 3.10 there is a schematic of the flow between tasks.

To be a real *self-ordering* FIFO, the values inside must be organized by some process, like the one used in windowed FIFO [6]. In this type of FIFO the producer or the consumer needs to acquire a window before write or read operations. During this time the values inside can be freely modified or ignored. When the operations are done there is a commitment; a *consume* in case of reading or a *release* in case of writing, and the consumer can, only now, remove the values or the producer can write more data in the buffer.

Like was said before, the overhead associated to the FIFOs accesses is quite evident so, if we had modified the software routines to implement a MIFO, the results would probably be even worse and useless. The only purpose would be to see if it is working to then build onto hardware as an IP core or even as a Nios II custom instruction. However, in this case, all the FIFOs have already the indexes positively organized so there is no need to internally re-organize data. The goal is make the *Destination* task to receive the indexes sequentially, even when the workers run at different speeds. For that, the *Ordering* task just needs to analyze the current index and, if it is the expected one, just moves it to the $FIFO_{4,5}$. This way the *Destination* task will have always the indexes already ordered.

Table 3.6: Results for 1000 tokens processed with different buffer sizes and different times performed on the model described in figure 3.10

Test	**Workers Time (ms)**		**FIFO$_{2,4}$ Size**	**FIFO$_{3,4}$ Size**	**FIFO$_{4,5}$ Size**	**Average Time per Token (ms)**	**Theoretical Time (ms)**	**Total Time (ms)**
#1	5	10	6	6	9	3.36	3.33	3,355.57
#2	5	5	6	6	9	2.51	2.50	2,514.41
#3	1	10	6	6	9	2.86	0.91	2,865.99
#4	1	10	12	2	9	2.01	0.91	2,008.40
#5	1	10	12	2	1	2.01	0.91	2,008.19
#6	1	10	20	20	9	1.45	0.91	1,449.27
#7	1	10	20	20	1	1.45	0.91	1,449.27
#8	1	10	40	40	4	0.97	0.91	968.17
#9	1	10	50	50	4	0.97	0.91	968.09

In the table 3.6 is shown the results by varying the buffers size, and the average time of the workers. We have concluded that $FIFO_{4,5}$ has a little effect on the overall times but the size of the buffers is very important when workers run at different speeds.

To optimize the overall performance it is possible to iteratively change the size of each FIFO while measuring the global times. While the difference between two loops is bigger than a given value, its approaching for the optimal buffer size. Otherwise, if a given threshold is not specified, the buffers will not have boundaries.

For two workers, it is curious to see that when the times of the workers are similar, the buffers do not need to be very big. When is the opposite happens, and workers have considerable different speeds among themselves, then the size of the buffer that is connected to the faster worker, needs to be at least three times bigger than the number of times that the fastest task can execute during one execution of the slowest one.

3.5 Concluding Remarks

In this chapter we have presented a topology based on the framework used by Google, *MapReduce*, which has a very useful mechanism to handle an huge amount of data in a shorter time. It is based in the splitting of data on several independent computers during the mapping phase, and when they finish the processing, the results are joined together during the reducing phase. We used this general notion in a typical pipelined synchronous data-flow to parallelize one particular stage, trying this way to reduce the WCET.

However, in our first experiments, we did not send the data to be processed through the buffers. Instead, we only send the order (as a 32-bit message) since the data was already inside the workers. This way the overall picture was simplified since our research was about the synchronization and communication mechanisms between tasks and not about the real computation. If one certain application can be depicted as a data-flow model, then it is possible to map it on a MPSoC platform.

We presented a synchronization policy based on the sending of a message to the workers and, when they finish the tasks, they produce an interruption to the mapping task and the results are sent, through buffers, to the reduce task. The mapping phase can be used as a firing mechanism of a centralized scheduler (mapped on a different processor) since it has a low overhead. Also, the Mailbox and IRQ scheme was used to evaluate a truly random seed that can be used to fill the first content of the shift register used in a random generator.

We also addressed to the problem of synchronization and communication of tasks in different processors using only FIFOs and between tasks inside the same processor using a real-time operative system, although we have skipped the real-time semantics and focused on the functionality and performance instead. However, this can be later extended for real-time applications since the timings and mechanisms between tasks are described. If the application can be modeled as a graph, where each node has a processing component, it is easier to transcribe it to the hardware. Furthermore, if the WCET of each node is known, a good scheduling policy can be applied in order to respect the real-time constraints and requirements, while at same time it allows to reduce the cost of the system.

Chapter 4

Monte Carlo π Generator

In chapter 3 there is an explanation of how the synchronization is performed between several tasks, running or not in different processors. In this chapter a case study with a Monte Carlo application is discussed using the same methodology, also with the goal to test the pseudo-random generator built as Nios II custom instruction.

4.1 Introduction

The goal is to show the speed-up of parallel computing on MPSoCs when compared to a single core SoC. Monte Carlo π generator is not intended to find the best possible value of this universal mathematical constant. In fact, it is useful to test the "randomness" of a given random generator to see if it is good enough so it can be later used in multiple different devices. Also, this experiment suits the purpose to test and evaluate the platform exposed in the previous chapter.

> "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.", John von Neumann[1] [28]

This chapter starts with an explanation about the Monte Carlo method, followed by the description of the pseudo-random generator used to test the method and the synchronization between tasks. Next, one simple way to compute π is described with the respective results.

4.1.1 Monte Carlo Method

This method describes a large and widely-used class of approaches of computational algorithms that rely on repeated random sampling to compute their results. They are specially useful in simulations of systems with a large number of coupled degrees of freedom, such as fluids, for instance. Thus, they are used when it is unfeasible or impossible to compute an exact result with

[1] John von Neumann was one of the most brilliant mathematicians from 20th century. He is mentioned here due to his contribution for Monte Carlo method during his work on hydrogen bomb.

a deterministic algorithm [29]. The term “Monte Carlo” appeared during 1940’s by the physicists working on nuclear weapon projects in the Los Alamos National Laboratory because they needed a method to test the radiation shielding and the distance that neutrons would likely travel through various materials. Despite they had most of the data, the analytic tests were not enough, so they modelled the problem using chance. Because those experiments were secret, they had to coin the research as “Monte Carlo” in a reference of Monte Carlo Casino in Monaco [30].

However, there is no single method but a relative common pattern followed by most of the applications:

1. Definition of the domain of possible inputs;
2. Generation of inputs randomly from the domain using a certain specified probability distribution;
3. Perform a deterministic computation using the inputs;
4. Aggregate the results of the individual computations into the final result.

There are several applications using this method. Most of them share the fact that they need *chance* to reach valid and non-deterministic results. In engineering, Monte Carlo methods are used for sensitivity analysis and quantitative probabilistic analysis because typical process simulations are interactive, and show a non-linear behaviour. They are also widely used in physical sciences, design and visuals, finance and business, telecommunications and games. Also in mathematics they play a key role on solving several problems like integration and optimization.

The application that was used to test the communications between tasks follows this pattern, and it will be explained in the following section.

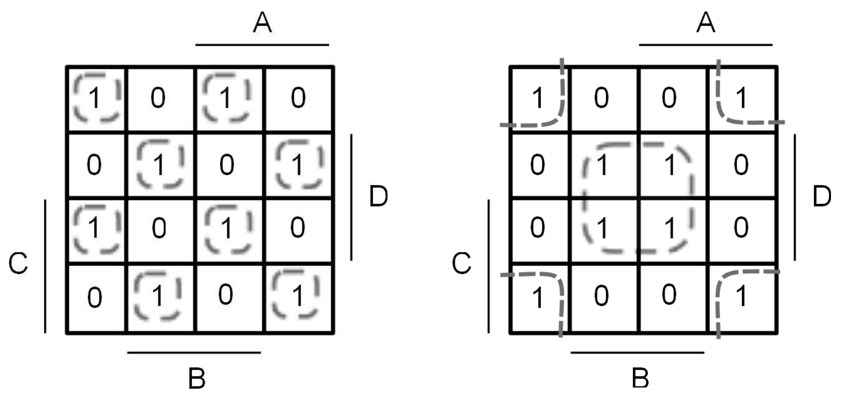
4.1.2 Pseudo-Random Generator

In chapter 3, a pseudo-random generator was used to change the time of each worker in order to see the functionality of the synchronization by buffers when only the WCET is known. A table with 1000 results was generated *before* the actual computation because the software version of the linear feedback shift register was too slow, even with the software improvements made to speed-up the generation of new random values.

The Monte Carlo method is precisely based on the generation of a lot of random values to perform several kinds of different simulations, so the need to build in hardware was evident. Furthermore, the initial 7-bit software version could only deliver up to 127 values, which is a too short range to be used in the application that we recreate to test the multiprocessor platform.

In the figure 4.1(a) there is the original Karnaugh map used on the software pseudo-random generator with the logic function $F = \bar{A}.\bar{B}.\bar{C}.\bar{D} + A.B.\bar{C}.\bar{D} + \bar{A}.B.\bar{C}.D + A.\bar{B}.\bar{C}.D + \bar{A}.\bar{B}.C.D + A.B.C.D + \bar{A}.B.C.\bar{D} + A.\bar{B}.C.\bar{D}$.

The probability of having a logical 1 or 0 is 50% since they are equality distributed in the truth table. However, the direct translation to hardware would require eight 4-input *AND* gates



(a) Software version (b) Hardware version

Figure 4.1: Simplification of Karnaugh map to reduce the amount of logic gates

and one 8-input *OR* gate which means a lot of logic gates. The propagation delay and the size required makes it a bad choice, so the truth table was reorganized to have the same probability but with only two groups of bits, as seen in 4.1(b). This way the logic function is smaller while maintaining the probability. The logic function is now $F = B.D + \bar{B}.\bar{D}$ and the gates used can be seen in the figure 4.2.

The first hardware version produced only 8-bit values, and the shift register was only 16-bit wide. However, all the connections to the Nios II were tested and it has been proved that the custom instructions feature is really useful. It was written in Verilog and in the section 2.2.2 is described the way how the signals are defined.

It only requires two clock cycles to drive a new result to the output port and all the multiplications needed by the software version are this way avoided. The speed-up was enormous as seen in the table 4.1.

Table 4.1: Performance comparison between the custom instruction and the software pseudo-random number generators for 10,000 values

	Total Time	**Time per Value**
Software 7-bit PRNG	58,553.98ms	5.85ms
Hardware 8-bit PRNG	32.09ms	3.21μs
Hardware 32-bit PRNG	30.41ms	3.04μs

After the 8-bit version was completed, the width of both shift register and the output value was increased to 64-bit and 32-bit, respectively. The 32-bit version is even faster because all the port width is used and there is no need to perform extra shifting operations to reduce the size to 8-bit.

This hardware module is used as a random generator of Cartesian points in order to easily evaluate the value of π through the Monte Carlo method described before. It is also useful to test the quality of the pseudo-random generator where the probability of hitting a certain value should be about the same within all range.

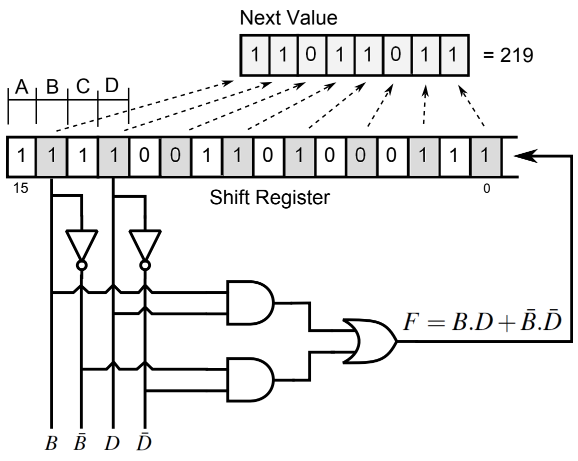


Figure 4.2: Logical implementation of a Linear Feedback Shift Register

4.2 Software Application

This is a simple embarrassingly parallel application that was initially computationally intensive with minimal communication between processors, and with a minimal I/O operations. Later the communication was increased by using a centralized random generator in order to extend the content introduced in chapter 3. The idea is to know the area of the circle and the square by using random shots inside them, in order to calculate π as seen in figure 4.3, and to further analyse the synchronization. The equations to evaluate the constant are shown below and the only important constraint is regarding the radius of the circle that has to be twice the width of the square.

The flow of the program is the following and it is possible to see how close it is of the "typical" Monte Carlo method:

1. Inscribe a circle in the square;
2. Randomly generate points in the square;
3. Determine the number of points in the square that are also in the circle;
4. Divide the number of points in the circle by the number of points in the square;
5. The constant π is approximately four times that value.

If the random generator have good properties, and the probability to hit certain number is about the same for them all, the more points generated leads to a better approximation.

$$\begin{aligned} CircleArea: \quad A_c &= \pi r^2 && (4.1)\\ SquareArea: \quad A_{sq} &= (2r)^2, && (4.2)\\ &= 4r^2. && (4.3) \end{aligned}$$

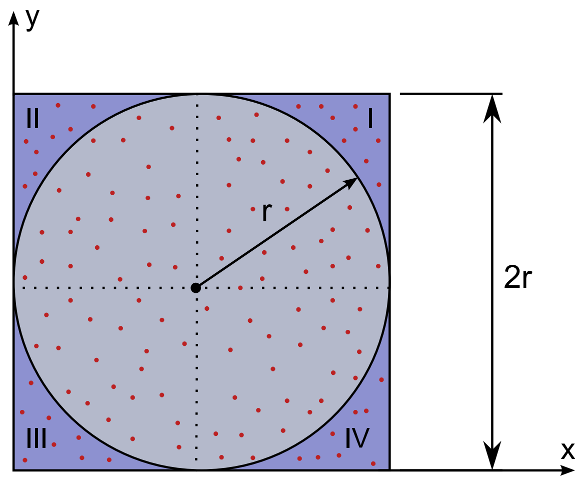


Figure 4.3: How are the random numbers used to evaluate π.

$$\begin{aligned} Pi: \quad \frac{A_c}{A_{sq}} &= \frac{\pi r^2}{4r^2}, && (4.4) \\ &= \frac{\pi}{4} \Leftrightarrow && (4.5) \\ \Leftrightarrow \pi &= 4 \cdot \frac{A_c}{A_{sq}}. && (4.6) \end{aligned}$$

It was built in the platform presented in chapter 2 and, because of the fact that the workers tasks were faster than the scheduler, it was required to add one more fast Nios II soft-core processor with TCM in order to get more results. However, the first results were taken using a 3 processors platform, where the two fast Nios II with TCM are used as workers, and the third one is both the interface and the scheduler.

The first application was intended to find the best possible performance, using the available resources, in order to have a better insight of how the communications between processors might decrease the benefits of using parallel processors when not well designed. This way, our purpose was trying to reach this value with more communication between tasks so we could find better ways to reduce the overhead caused by the shared memory.

Both workers received the custom instruction module responsible to generate random values in order to have a distributed random numbers generation. The master processor needed only to send through the buffers the number of points that the workers should process each time. This way the communications are kept to the minimum and the speed is the highest possible. The buffers size is only four slots and they are always full. However, the master core has to wait either way so, even when the buffer was bigger, the difference is not significant. For the case where worker 1 process 1,000 pairs each time and worker 2 process 10,000 pairs, 10 million pairs where processed in 12.69 seconds which means that each pair took about 1.27μs and π was 3.141498. For 50K pairs, the most typical amount used and, from now on, the reference value, the time was 63.50ms with the following result: 3.145584.

Using a centralized way to generate and distribute the random Cartesian points the performance is worst due to the higher communication between processors but, on the other hand, several new tests can be made to evaluate better ways to optimize communication, one of the goals of this thesis.

Table 4.2: Estimation of π after N iterations

No. of Iterations	π Estimation	Error
50K	3.08088	1.93%
100K	3.10064	1.30%
150K	3.11795	0.75%
200K	3.12838	0.42%
250K	3.13318	0.06%
500K	3.14134	0.01%
1000K	3.13966	0.06%
2000K	3.14278	0.04%
100M	3.14159	$<0.001\%$

Initially, two sequential versions were made using only one processor to see if everything was working properly since there is the need to process 64-bit variables using a 32-bit RISC soft-processor. No operative system was used at this point, it was purely raw computation. In the first version, the size of the square was one and, when divided by 2^{31} (or 2,147,483,648) small segments, it has required the use of double precision floating point operations, which consumes much more time and they are performed in software. The second one, which was later evaluated to work in parallel, only uses integers. This way the computation is faster using the same hardware. Thus, the size of the square is now 2^{31} and the 32-bit random values are shifted one bit to became directly usable. This way there is no need to perform the division, hence resulting in a significant performance improvement.

Table 4.3: Timing results for the sequential versions for 50K Cartesian points

Size of the Square	Time per Point	Total Time
1	625.35μs	31,267.65ms
2^{31}	12.34μs	617.05ms

For both version the results for the π are the same since the seed was not modified; {32'd43608; 32'd43608} [2]. In the table 4.2, the values for different number of iterations are presented. It is important to note that the accuracy depends on the "randomness" of the generator, hence the fact that sometimes with more points the result has more error. In the table 4.3, the timing results are exposed for these two sequential versions. It is evident the speed improvement to perform exactly the same computation when the double precision operations can be avoided.

[2]This is the HDL (hardware description language) way to represent a concatenation of two 32-bit values. {A;B} is the concatenation of A with B, the 32' means that the value is 32-bit wide and the *d* is for the compiler to know that the value is written as decimal.

Also, these two centralized versions did not required the usage of buffers and the best one was 10 times slower than the fastest possible version discussed before. This was due to the fact that it was not parallel and, besides, it was running on the slowest processor that uses SDRAM to store information.

Next, the parallel version using the method with less division operations is addressed. It uses the flow depicted on the figure 3.2 in the previous chapter, where the abstract tokens were replaced by the coordinates of the points and the results, if the hit was inside or outside the circle.

For 50K points, where 25K was sent to worker 1 and 25K to worker 2, the total time was 4.21s, or 84.17μs/pair, which is almost 8 times worst than the sequential version *without* buffers. Once again, it was patent the overhead due to the access to the shared memory. The goal was now to decrease this value toward the "optimal", and impossible to reach, 1.27μs/pair with a clock frequency of 50MHz. With the compiling directive "-O3" that can be used to optimize the speed and the size of the program, the performance was slightly better: 81.11μs/point.

The next step was to optimize the code to reduce the number of accesses to the shared memory. The result is depicted in the subsection 2.4.3, regarding the software FIFOs. Also, semaphores were used to switch tasks inside the operative system in order to distribute and process the work among the workers, as described on chapter 3. The overhead was still evident but we experienced a improvement of about 37%. Now, with exactly same conditions and same hardware, it took 51.09μs to process each pair of coordinates.

At this point, the question of why does the sequential version running on a slower processor was way faster than the parallel version still continues. Most of the processing was performed on faster processors with coupled memories. To answer that, a test was made by adding the external buffers to the sequential version. This way it was possible to avoid any bottleneck effect. In fact, a lot of traffic on the bus is produced by the workers running on the fast processors, since they are always trying to access to the buffers. The results are presented in the table 4.4.

Table 4.4: Timing results for several versions after generating 50K Cartesian points

	Version	**Time per Point**	**Total Time**
Single Core	Manually Optimized [3]	12.34μs	616.98ms
	With Buffers	79.19μs	3,959.82ms
MPSoC	Using 1 Independent Worker	64.25μs	3,212.47ms
	Using 2 Independent Workers	51.09μs	2,554.97ms

It is possible to see that the overhead associated to the addition of the external FIFOs is 67μs/point for the standard Nios II with SDRAM. The tests to verify the current status of the FIFOs (if they have free space or tokens to be consumed) takes in average 24μs. The idea was to keep checking without being so intrusive but, without an interruption generated by the buffer or by the workers, it is not possible. However, it has proved that the non sequential access to the Avalon

[3]The sequential manually optimized version does not use any FIFOs to temporary store values, hence the bigger performance. The version with buffers is the one that is meant to be compared to the parallel ones.

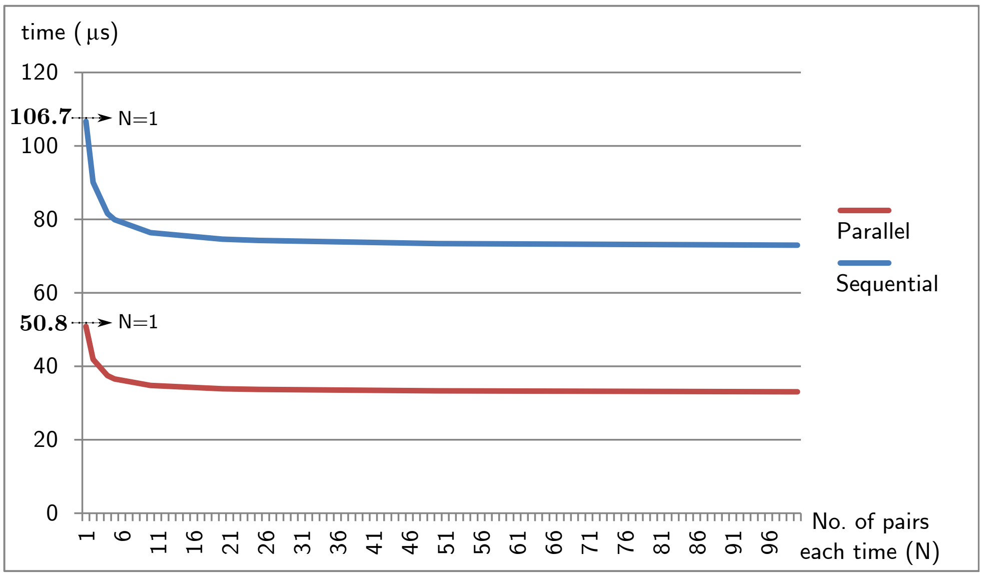


Figure 4.4: Performance improvement by sending N pairs of coordinates each time

Switch Fabric decreases the performance, specially when using the standard version of Nios II with SDRAM as its storage place for data and instructions.

By default, SOPC Builder builds a partial crossbar fabric that connects master and slave components. This is ideal for ASIC or FPGA structures due to the fact that it requires fewer resources and allows parallelization. This way the biggest disadvantages of having only a bus or a full crossbar switch architecture are avoided. The bus topology can achieve relatively high clock frequencies at the expense of little or no concurrency. In fact, a shared bus architecture can lead to a significant performance penalty in systems with many bus masters because all masters compete for the access to the shared bus rather than a particular slave device. On the other hand, in the full crossbar switch fabric, it is possible to have multiple concurrent transactions. Furthermore, it is flexible and provide high throughput over a higher hardware cost as it grows exponentially as more masters and slaves are added [20].

The partial crossbar fabric is automatically generated by SOPC Builder during the generation of the system's description. It has the advantages of both the bus and full crossbar topologies. Furthermore, since the streaming interface was not used because it did not fit in our goals, the times when accessing to the external memory proved to be too slow and have limited the use of this approach.

One way to overcome this problem is to send more data each time to the workers in order to force a more sequential access. Therefore, instead of sending only one (x, y) random pair to the worker, the size of the FIFOs was increased to store more values. We have noticed that the overall time was decreasing as more pairs were sent each time as seen in the figure 4.4. The parallel version is exactly the same as the sequential one, only with the difference that two workers are mapped on different processors. The sequential version uses external buffers to store the values in order to mimic (with the associated overhead) the parallel version, however using only one processor. It is interesting to see the initial exponential decrease toward the ideal limit fixed by the amount of computation on the slower processor. Also, in the sequential version, care was taken to avoid any bottleneck effect that could effect the experiments. Hence, the Avalon fabric, even when is not using any streaming modules, works better for sequential accesses.

A final test was made to check the performance speedup achievable by adding one more fast Nios II processor to the platform to act as the scheduler, instead of the slow standard Nios II that uses SDRAM. The architecture is shown in the figure 2.10 and it has 4 processors. 50K pairs of random Cartesian points were again processed using two fast workers with coupled memory and we could avoid the limit imposed by the SDRAM. However, when we tried to run the tests with the RTOS synchronization, we found that the available on-chip memory was not enough. The operative system needed to be compiled with a minimal set of services in order to reduce considerably the size.

The slower processor was still available to act as the bridge between the computer and the FPGA, however performing a passive job and without interfering in the application. It just reads the results *after* 50K points have been processed.

The results are available in the table 4.5, where it is possible to see the speedup when compared to the platform where the master processor uses SDRAM for data and instructions storage place. Similarly, when the master sends more than one point each time to the workers, the performance is better due to the lower rate of accesses to the bus by different masters.

Table 4.5: Performance speedup by sending bursts of points each time

No. of Points	**Single Core**		**MPSoC**	
	SDRAM	**TCM**	**Master uses SDRAM**	**Completely On-Chip**
1	106.69μs	24.98μs	51.09μs	12.53μs
2	90.09μs	22.23μs	41.88μs	11.12μs
...	...	...	...	....
50	73.42μs	19.21μs	33.29μs	9.43μs

4.3 Concluding Remarks

In this chapter, a case study using the data-flow model described in chapter 3 was presented with a simple application based on the Monte Carlo method. It estimates the value of π by generating a lot of random points. Furthermore, we have implemented a pseudo-random generator as a Nios II custom instruction.

With this application we have analysed the overhead of communications and several issues that are inherent to multiprocessor systems-on-chip, such as the interconnect fabric limitations, the trade-offs between high-speed on-chip memories versus the external memory with higher capacity, the non-sequential access to the buffers, and the triggering of tasks on different processors.

Chapter 5

Conclusions and Future Work

In the course of this work, several approaches to evaluate and test communications in MPSoCs were addressed in order to build an efficient workable model that could be easily adapted to accommodate different purposes. For that, different synchronization policies were made using dynamic scheduling.

We have presented several ideas to prototype streaming applications on FPGA based MPSoCs using the *MapReduce* inspiration to parallelize tasks or stages. This was accomplished in a simplified manner in order to analyse the overheads and performance, and thus, optimizing the flow.

Knowing the times required by the transactions of data and the overheads associated, it is possible to build a better real-time system using a multi-processor environment.

Some heterogeneous multi-processor systems were built in the Stratix II FPGA chip using SOPC Builder with tightly coupled memories. The communication between all the components using the Avalon Switch Fabric was made in software and can be later masked so the designers only need to be concerned about the division of the application and the mapping of the several routines onto the processors or in dedicated hardware modules. It is also relatively easy to have a better insight of the several blocks and layers used in order to customize them according to the needs.

Also, it was proved the enormous capabilities of the Nios II soft-core processors by adding custom instructions directly to the ALU, and the advantages of using the default topology of the interconnect fabric. The partial crossbar switch is a good way to connect all the modules in a embedded system. By using the slave-side arbitration scheme, the bottleneck is not a major problem since it gathers the full crossbar switch (where one input leads to one dedicated output) and the traditional bus (where the speeds can be higher but there is the need to have only a single master each time). Of course that, in more complex systems with a huge flow of information in the buses, different approaches must be used to avoid bottleneck. However, this work can be easily extended to optimize the communications of a sequential pipelined synchronous data-flow model or in a *MapReduce* way that can be reorganized as a central scheduler (or just *Map*).

The communication entities were tokens (or abstract units of data) but it can be later extended to process packets. By using special nodes that divide work and special nodes that join the results it is possible to extend and model this work in order to have switches that route the packets toward the destination, a bit similar to what is done in NoCs (Network-on-Chip).

Besides the communication between nodes through buffers, a test mechanism based on the randomness was addressed. A Nios II custom instruction with a truly random seed was evaluated. It uses a linear feedback shift register and can be very useful to find potential synchronization problems in multi-core platforms such as race conditions, deadlocks, starvation, bottleneck, etc.

5.1 Future Work

This work can be extended in many directions. The conclusions and results could have been more interesting if a hardware FIFO was used since one of the biggest backdraw was the low performance of the FIFO written in software, because of the slow access to on-chip memory by the CPU that has used the external SDRAM as data and instructions storage.

One easy way to improve communications between processors inside the FPGA is by adding custom instructions to the Nios II that handles all the external data communications and synchronization. This way, it is possible to avoid (or optimize) the Avalon Switch Fabric or even create a new customized and dedicated fabric for interconnection. Also, Altera allows the use of several dedicated slave interfaces for streaming purposes that can be used to speed up important critical data transactions.

This special custom instruction may connect to a separate network management module, which is a central core that is connected to all nodes, that can be processors or some other devices. This way the latency and speed of communications can be optimized according to the application.

Experiments were made by using a communication assistant (CA) module to reduce the overhead of the data transmissions but it did not showed any benefits because the processor accessed the on-chip memory exactly the same way as before, using Avalon Switch Fabric and without cache. However, in this case there was no shared memory involved, neither race conditions, since that each processor had his own local and private space. The CA was built to access the local space of one processor, read the contents of the FIFO and copy the values to the local space of another processor. This procedure was automatic, running on background and the goal was to minimize the overhead and the bottleneck associated to the communication between processors. Communication assistant is always updating the buffers in two different processors. One idea is to have the buffers directly attached to a special Nios II custom instruction. This way, it is possible to have a significant speeding up of data communication between two or more processing devices that can be also allied to a cache-coherency protocol.

In a streaming application embedded in a MPSoC, using a synchronous data-flow way to distribute work among the processors, it is possible to analyse fragments of code that can be embarrassingly parallel in order to better split the flow (and total computation) among processors.

Later the output is treated and, through a buffer, sent to the following stage. By forking pipeline stages, we can have trade-offs between more CPUs, performance gains and buffer requirements.

As a final remark, this was meant to be a basis of something bigger. An automated process to analyse, separate and organize parallel work in a higher level still needs to be evaluated to help programmers and designers to simplify thoughts. In fact, we are currently seeing a technological improvement that it is not being followed by the programmers. Methodologies are being created but there is still missing a way to switch from ordinary sequential way of thinking to a parallel one but chances are that it will slowly appear with the contributions of a lot of people.

References

[1] Benaoumeur Senouci, Abdellah.M Kouadri.M, Frédéric Rousseau, and Frédéric Petrot. Multi-CPU/FPGA platform based heterogeneous multiprocessor prototyping: New challenges for embedded software designers. *Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium*, pages 41–47, 2008.

[2] Terry Tao Ye, Luca Benini, and Giovanni de Micheli. Packetized on-chip interconnect communication analysis for MPSoC. *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 344–349, 2003.

[3] Mirko Loghi, Federico Angiolini, Davide Bertozzi, and Luca Benini. Analyzing on-chip communication in a MPSoC environment. *Design, Automation and Test in Europe Conference and Exhibition, 2004*, pages 752–757 Vol.2, 2004.

[4] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. *Proc. IFIP Congress, Stockholm, Sweden*, pages 471–475, 1974.

[5] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE, vol. 75, no.9*, pages 1235–1245, September 1987.

[6] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed FIFOs. *7th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'09)*, pages 35–44, October 2009.

[7] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. In *IEEE Transactions on Computers*, pages 24–35, January 1987.

[8] Jun Zhu, Ingo Sander, and Axel Jantsch. Energy efficient streaming applications with guaranteed throughput on MPSoCs. In *Proceedings of the 7th ACM international conference on Embedded software (EMSOFT '08)*, pages 119–128, October 2008.

[9] R. Govindarajan, Guang R. Gao, and Palash Desai. Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks. *Journal of VLSI Signal Processing*, 31(3):207–229, July 2002.

[10] Sander Stuijk, Marc Geilen, and Twan Basten. Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Design Automation Conferrence, Proc. ACM*, pages 899–904, July 2006.

[11] Jun Zhu, Ingo Sander, and Axel Jantsch. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In *Proceedings of Design Automation and Test in Europe (DATE '09)*, April 2009.

[12] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. *Proceedings of 13th International Symposium on High-Performance Computer Architecture (HPCA)*, 2007.

[13] Chunhua Chen, Gaoming Du, Duoli Zhang, Yukun Song, and Ning Hou. Communication synchronous scheme for MPSoC. In *Anti-Counterfeiting Security and Identification in Communication (ASID '10)*, pages 310–313, July 2010.

[14] Altera Corporation. Nios Development Board - Reference Manual, Stratix II Edition, 2004.

[15] Altera Corporation. Stratix II Device Family Data Sheet, 2007.

[16] Altera Corporation. Nios II Processor Reference Handbook, 2008.

[17] Altera Corporation. Nios II Custom Instruction - User Guide, 2008.

[18] Altera Corporation. Altera Avalon Interface Specifications, 2008.

[19] Altera Corporation. Quartus II Handbook: Version 10.0 - Volume 4: SOPC Builder, 2010.

[20] Altera Corporation. Embedded Design Handbook - Avalon Memory-Mapped Design Optimizations, 2008.

[21] Altera Corporation. Quartus II Handbook: Version 8.1 - Volume 5: Embedded Peripherals, 2008.

[22] Altera Corporation. Using Tightly Coupled Memory with the Nios II Processor, 2009.

[23] Frank Schirrmeister. Multi-core processors: Fundamentals, trends, and challenges. *Embedded Systems Conference (ESC351)*, 2007.

[24] Jean J. Labrosse. *MicroC/OS-II - The Real-Time Kernel*. CMP Books, Second edition, 2002.

[25] Micrium. microC/OS-II product datasheet.

[26] OSCI Open SystemC Initiative. System C, October 2010. http://www.systemc.org/home/.

[27] Hajer Chtioui, Rabie Ben Atitallah, Smail Niar, Jean-Luc Dekeyser, and Mohamed Abid. A dynamic hybrid cache coherency protocol for shared-memory MPSoC. *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 3–10, August 2009.

[28] John von Neumann. Various techniques used in connection with random digits. *Applied Mathematics Series, no. 12*, pages 36–38, 1951.

[29] Computational Science Education Project. Introduction to Monte Carlo methods, 1995. Retrieved from http://www.phy.ornl.gov/csep/CSEP/MC/MC.html.

[30] riskglossary.com Glossary, Encyclopedia & Resource Locator. Monte Carlo Method. Retrieved from http://www.riskglossary.com/link/monte_carlo_method.htm.